Abschätzung der Leistungssteigerung durch 3D-DRAM und Evaluation einer lokalitätsbasierten Architektur

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

von

Dipl.-Ing. Alex Schönberger

Geboren am 25.10.1984 in Jenotajewka / Russland

Referent:	Prof. Dr.-Ing. Klaus Hofmann
Korreferent:	Prof. Dr.-Ing. Dr. h. c. Jürgen Becker

Tag der Einreichung:	14.04.2016
Tag der mündlichen Prüfung:	10.06.2016

Darmstadt 2016
D17

Erklärung laut §9 PromO

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, den 26. Juni 2016

Alex Schönberger

Zusammenfassung

Das Speichersystem moderner Rechner ist in den meisten Fällen aus mehreren heterogenen Komponenten aufgebaut. Dabei bilden diese Komponenten eine Hierarchie, deren Ebenen sich durch wachsende Speicherkapazität und gleichzeitig zunehmende Zugriffszeit unterscheiden. Eine dieser Ebenen ist DRAM. Dieser Speicher ist in der Lage mehrere Gigabytes an Informationen aufzunehmen, wobei seine Leistungsfähigkeit gegenüber der CPU um bis zu vier Größenordnungen geringer ist. Diese Differenz in der Leistungsfähigkeit wird als *„memory wall“* bezeichnet und erfordert zusätzliche Maßnahmen im Gesamtsystem. Ein weiterer Zweig der Technologie der integrierten Schaltungen stellt die Stapeltechnik dar. Dabei können mehrere Halbleiterschichten übereinander gestapelt werden. Fortschritte in dieser Technologie, die es erlauben, Zwischenverbindungen innerhalb der Schichten in großer Zahl und an beliebiger Stelle zu platzieren, könnten eine Alternative für diese Maßnahmen bilden und so die Leistungsfähigkeit des Gesamtsystems steigern. Inwiefern es möglich ist und wie stark die Verbesserungen sein könnten, ist Gegenstand der Untersuchungen dieser Arbeit.

Die entscheidende Größe für die Leistungsfähigkeit eines Systems ist die Ausführungszeit einer Applikation. Dabei benötigt diese Ausführung eine bestimmte Anzahl von Taktzyklen. Wenn jeder Speicherzugriff innerhalb eines Taktes ausgeführt werden kann, dann liegt ein idealer Speicher vor und die benötigte Ausführungszeit stellt eine Obergrenze für mögliche Verbesserungen am Speichersystem dar. Innerhalb der Speicherhierarchie bildet die oberste Ebene mit der geringsten Kapazität das Verhalten eines idealen Speichers ab. An diesen Verhältnissen hat sich seit Beginn der Zunahme der Integrationsdichte von digitalen Schaltungen nichts Grundlegendes verändert. Der Schlüssel für den Erfolg dieser Lösung liegt in einer Eigenschaft, wie die CPU den Speicher während der Ausführung nutzt. Manche Instruktionen und Daten werden im Vergleich zu anderen sehr viel häufiger gebraucht. Diesen Zusammenhang, besser bekannt als Lokalitätsprinzip, hatte Denning bereits 1968 beschrieben und damit den Weg für den erfolgreichen Einsatz vom Pufferspeicher geebnet.

Der physikalische Aufbau eines DRAM bietet durchaus Ansätze, um die Leistungsfähigkeit zu steigern, es sind vielmehr wirtschaftliche Aspekte, die dieser Entwicklung im Weg stehen. Zudem führt das Lokalitätsprinzip dazu, dass diese Steigerungen nur im geringen Maße das Gesamtsystem beeinflussen. Der untersuchte Lösungsansatz dieser Arbeit kombiniert diese beiden Erkenntnisse. Die Stapeltechnik erlaubt es die DRAM-Architektur um eine Schicht zu ergänzen, die einerseits auf Latenz optimiert ist und andererseits häufig genutzte Daten enthält. Die Platzierung erfolgt per Software.

Für die Untersuchungen wird ein allgemeines, auf keine speziellen Aufgaben zugeschnittenes System verwendet. Es werden sowohl Einkern- als auch Mehrkernarchitekturen betrachtet. Als Testapplikationen werden Implementierungen von unterschiedlichen Kompressionsalgorithmen verwendet. Für die Ausführung werden sowohl die Eingabedaten als auch ihre Menge variiert, um unter anderem den Einfluss des Pufferspeichers zu erkennen und aus der Untersuchung möglichst herauszunehmen. Darüber hinaus wird das Potential der Leistungssteigerung durch Stapeltechnik mittels künstlicher Manipulation der Lokalität geschätzt. Als ein Gegenbeweis wird zudem der volumenbasierte Ansatz ausgewertet.

Die Untersuchungen zeigen, dass die Stapeltechnik durchaus in der Lage ist, die Leistungsfähigkeit des Systems zu steigern. Die Vorteile dieser Technik liegen aber nicht primär in höherer Leistung, da die *„memory wall“* für das Gesamtsystem nur eine geringe Rolle spielt. Der vorgeschlagene Ansatz zeigt bessere Resultate als Vergleichsmessungen, der Grad der Verbesserung ist aber stark applikationsabhängig und dessen Auswirkungen auf das Gesamtsystem hängen von der Taktfrequenz ab.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Die rasante Entwicklung der Technologie seit der industriellen Revolution hat das Leben der Menschen so tiefgreifend verändert, dass sie sich nur ansatzweise mit anderen historischen Umbrüchen vergleichen lässt. Diese Entwicklung speist sich aus einer Fülle weitverzweigter Stränge. Einer davon ist der Vormarsch digitaler Datenverarbeitungssysteme. Dieser Zweig der Elektrotechnik durchzieht den Alltag mit immer neuen Lösungen. Nach Schätzungen von Hilbert und López werden seit Anfang der 2000er Jahre nahezu alle Informationen digital gespeichert [Hilbert und López, 2011], wobei dieser Anteil 10 Jahre davor noch unter 5 % lag. Über den weiteren Verlauf der technologischen Entwicklung lässt sich nur spekulieren. Momentan stellen digitale Systeme weiterhin eine wichtige Komponente der verwendeten Technologien dar.

Aus der Benennung lässt sich bereits ableiten, dass solche Systeme digitale Daten verarbeiten. Diese Daten werden dem System zugeführt. Daraufhin erfolgt ein Transformationsprozess und die veränderten Daten verlassen das System. Während der Verarbeitung müssen die Daten gespeichert werden. Damit sind alle Komponenten eines digitalen Datenverarbeitungssystems benannt: eine Ein- und Ausgabeeinheit, eine Verarbeitungs- und Speichereinheit [Ganzhorn und Walter, 1975]. Der Erfolg solcher Systeme ist unter anderem darin begründet, dass sie in der Lage sind, bestimmte Aufgaben mit einer Geschwindigkeit zu lösen, die keine vorherige Technologie auch nur annähernd erreicht hat. Diese Geschwindigkeit nimmt zu, wenn auch der treibende Faktor sich seit etwa Mitte des ersten Jahrzehnts des 21. Jahrhunderts änderte. Darauf wird später noch detailierter eingegangen.

Wenn man die Entwicklung der Leistungsfähigkeit der einzelnen Komponenten, insbesondere der Verarbeitungs- und Speicherkomponenten, über einen größeren Zeitrahmen betrachtet, lässt sich eine Besonderheit feststellen, die in der Abbildung 1.1 grafisch dargestellt ist.

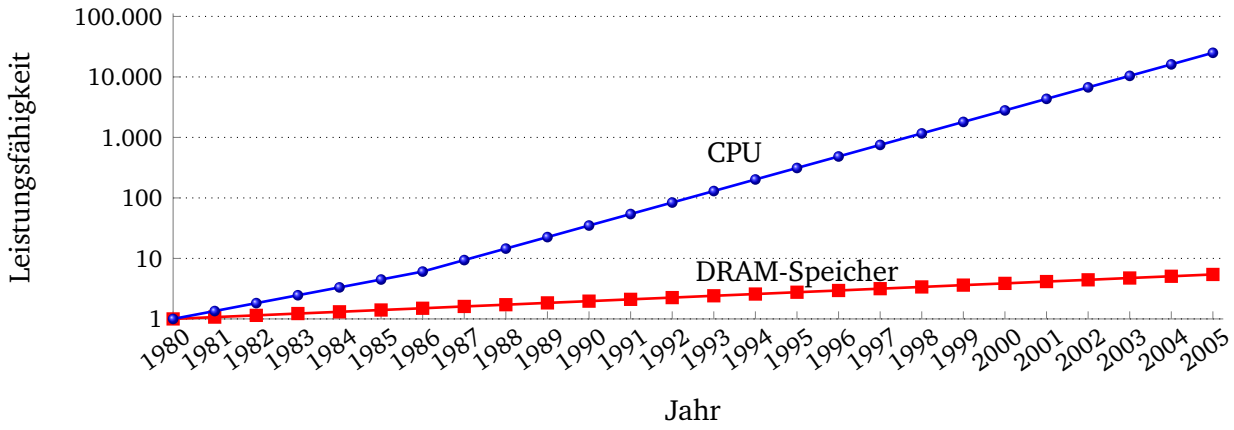


Abbildung 1.1: Auseinanderlaufen der Leistungsfähigkeit von CPU und DRAM-Speicher (hier anhand eines 64 kB DRAMs), bezogen auf das Basisjahr 1980. Für den Speicher wird eine durchschnittliche Zunahme von 7 % pro Jahr angenommen, für die CPU ein Faktor von 1,35 bis 1986 und 1,55 danach. Für DRAM-Speicher wird die Leistungsfähigkeit weitgehend auf die Latenz zurückgeführt. [Hennessy und Patterson, 2012]

Die Leistungsfähigkeit der Speicherkomponente (hier ein DRAM-Speicher) weist im Vergleich zur Verarbeitungskomponente (hier ***C**entral **P**rocessing **U**nit*) 2005 einen Unterschied von 4 Größenordnungen auf (die Y-Achse hat einen logarithmischen Maßstab). Ausgehend von dieser Betrachtung scheinen digitale Systeme ein enormes Potenzial zur Leistungssteigerung durch Schließung dieser Lücke zu besitzen. Eine signifikante Steigerung der Leistungsfähigkeit der Speicherkomponente sollte die Entwicklung somit enorm vorantreiben.

Einen weiteren Zweig des Fortschritts stellt die Weiterentwicklung der Stapeltechnik von heterogenen Systemen dar. Veränderte Herstellungsverfahren bieten eine Reihe von Vorteilen gegenüber den konventionellen Lösungen [Lienig et al., 2012], bei denen die gesamte Funktionalität nur innerhalb einer Lage realisiert werden muss. Diese Vorteile eröffnen wiederum Möglichkeiten, den DRAM-Speicher anders zu gestalten. Dies könnte signifikante Auswirkungen auf die Leistungsfähigkeit haben. Eine Betrachtung dieser Entwicklung im Zusammenhang mit der Speicherleistungslücke ist der Grundstein dieser Arbeit.

1.2 Fragestellungen dieser Arbeit

Eine Einschätzung der Auswirkungen neuer Technologien nur auf Basis vorhandener Untersuchungen ist bei nicht-linearen Systemen zu Fehleinschätzungen führen. Die veränderten Bedingungen können zuvor vernachlässigte Größen plötzlich zu einem dominanten Faktor werden lassen und umgekehrt. Eine Extrapolation, die sich im Nachhinein als zutreffend erweist, sticht vielleicht - wie die berühmte Vorhersage von Gordon Moore über die Zunahme der Integrationsdichte von elektronischen Schaltungen [Moore, 1965] - so hervor, weil solche Fälle eher selten sind. Ein neues Experiment, eine neue Bestandsaufnahme liefern weitaus verlässlichere Eckdaten in Bezug auf die eingangs erwähnte Einschätzung.

Daher lässt sich die erste Frage dieser Arbeit ganz grundlegend wie folgt formulieren:

Frage 1

Bietet die Stapeltechnik Möglichkeiten, die Lücke in der Leistungsfähigkeit zwischen einem DRAM-Speicher und einem Prozessor zu verringern oder sogar zu schließen?

Die zweite Frage baut auf der ersten auf:

Frage 2

Wie muss die DRAM-Architektur in der gestapelten Version verändert werden, um die Leistungsfähigkeit zu verbessern?

Die dritte Frage zielt auf eine mögliche Obergrenze für die Leistungssteigerung:

Frage 3

Wie hoch ist das Potenzial für die Leistungssteigerung durch die Stapelung von DRAM?

Die dritte Frage scheint auf den ersten Blick und insbesondere in Bezug auf die Grafik 1.1 schnell beantwortet zu sein. Doch ein direkter Vergleich der Leistungsfähigkeit eines Prozessors mit einem DRAM-Speicher sagt nichts darüber aus, wie sich die beiden Komponenten in einem Gesamtsystem verhalten würden. Diese Aussage ist nur unter Einbeziehung aller relevanten Bestandteile eines Gesamtsystems möglich. Diese Überlegung war die Basis für den Testaufbau dieser Arbeit.

1.3 Aufbau der Arbeit

Im weiteren Verlauf dieser Arbeit wird die Herangehensweise bei der Suche nach Antworten auf die dargestellen Fragen beschrieben. Das folgende Kapitel 2 gibt eine Einführung in das Speichersystem. Außerdem werden die theoretischen Erkenntnisse bzgl. der Speichernutzung präsentiert. Darüber hinaus wird die Struktur des Speichersystems erläutert. Die Innovationen der Stapeltechnik werden vorgestellt und es wird ein Überblick über die aktuellen Publikationen in diesem Bereich gegeben. Anschließend beschäftigt sich das Kapitel mit der Laufzeitmessung bei der Ausführung einer Applikation. Im letzten Abschnitt werden die zentralen Hypothesen der Arbeit aufgestellt.

Die Beschreibung des Testaufbaus ist auf zwei Kapitel aufgeteilt. Das Kapitel 3 konzentriert sich auf die Hardwarekomponenten. Der verwendete CPU-Kern und der Zusammenschluss zu einem Mehrkernsystem werden beschrieben. Der dritte Abschnitt des Kapitels geht auf die Kernkomponente der Untersuchung ein: das Speichersystem. Es wird sowohl das Referenzsystem als auch der Pufferspeicher und anschließend das untersuchte 3D-DRAM Model vorgestellt und erläutert.

Im Kapitel 4 wird ebenfalls die Testumgebung beschrieben, allerdings in Bezug auf die Software. Ein eigens für diese Arbeit entwickeltes Analysewerkzeug wird vorgestellt. Danach folgen die Testapplikationen. Diese lassen sich in real anwendbare und speziell für die Untersuchung entwickelte Implementierungen unterscheiden. Daneben werden die zusätzlich implementierte Betriebssystemfunktionalität sowie die unterstützende Softwareumgebung beschrieben.

Die nächsten Kapitel 5 und 6 präsentieren die Ergebnisse der Untersuchung. Die Diskussion der Messwerte erfolgt direkt in den Kapiteln, wobei die Hypothesen der Arbeit einen Rahmen für die Einordnung bilden.

Das letzte Kapitel 7 fasst die wichtigsten Erkenntnisse der Arbeit zusammen, nimmt Bezug auf die Hypothesen (die im Abschnitt 2.4 aufgestellt werden) und schließt die Arbeit mit einem Ausblick ab.

2 Speichersystem, Stand der Forschung und Herleitung der Hypothesen

2.1 Hauptspeicher und seine Subsysteme

In jedem digitalen Datenverarbeitungssystem lassen sich neben Datenein- und ausgabe zwei weitere Hauptkomponenten auffinden: eine Verarbeitungseinheit (Steuerteil und Rechnerwerk) und eine Datenaufbewahrungseinheit [Ganzhorn und Walter, 1975]. Beide Komponenten können vielfältige innere Strukturen aufweisen, über Module verfügen, die teilweise die Funktionalitäten der jeweils anderen Komponente übernehmen. Die Verbindung zwischen den beiden kann ebenfalls auf diverse Arten realisiert sein.

Solche Datenverarbeitungssysteme werden zudem über eine Reihe von Parametern charakterisiert. Neben Energieverbrauch und Kosten hat die Leistungsfähigkeit einen herausragenden Stellenwert. Dabei spielt die absolute Rechenzeit von Vergleichsprogrammen (*benchmarks*) eine entscheidende Rolle. Hier kommt es in erster Linie auf die beiden eingangs erwähnten Hauptkomponenten an, wobei bei modernen Systemen die Verarbeitungseinheit eine Grenze für die maximale Rechenzeit aufzeigt [Ferrari, 1978]. Bei der Datenaufbewahrungseinheit nimmt die Zugriffsgeschwindigkeit mit der Speicherkapazität ab und entscheidet damit über die tatsächliche Ausführungszeit.

Der Begriff „Datenaufbewahrungseinheit“ umfasst die gesamte Vielfalt der technischen Lösungen für die Speicherung digitaler Daten. In der Regel wird in modernen Systemen nicht nur eine Lösung verwendet, da jede Technologie im Hinblick auf bestimmte Aufgaben Vorteile aufweist, sodass spezielle Lösungen das Gesamtsystem verbessern. Diese Heterogenität kann aus der Applikationssicht in mehrere Abstraktionsbereiche unterteilt werden. Einer dieser Bereiche wird als **Hauptspeicher** bezeichnet und umfasst den gesamten Adressraum, auf den die Ausführungseinheit zugreifen kann [Jacob et al., 2007]. Die Stapeltechnik ist zwar nicht auf diesen Bereich beschränkt, wird jedoch im Rahmen dieser Arbeit speziell für diesen Teil des Speichersystems betrachtet.

2.1.1 Eine Speicherhierarchie

Die technologische Entwicklung hat über die letzten Jahrzehnte mehrere Realisierungsmöglichkeiten zum Speichern digitaler Daten hervorgebracht. Dabei lassen sich zwei Arten von Technologien unterscheiden: nichtflüchtige Speicher, die ihre Daten auch ohne die Energieversorgung beibehalten, und solche, die nur bei Aufrechterhaltung der Energiezufuhr eine Speicherfunktionalität bieten. In Bezug auf den Einsatzbereich der jeweiligen Technologie lassen sich die Grenzen nicht so klar ziehen. Im Allgemeinen werden die meist langsameren nichtflüchtigen Speicher zur Aufbewahrung von Daten genutzt, während die meist schnelleren flüchtigen Speicher bei der Ausführung einer Applikation eine wichtigere Rolle spielen. In dieser Arbeit geht es um die Leistungsfähigkeit eines Systems und somit um Ausführungszeiten von Applikationen. Daher wird im weiteren Verlauf auf die Speicheraufbewahrungssysteme nicht weiter eingegangen. Diese Systeme gehören nur bei speziellen Realisierungen zum Hauptspeicher.

Während der Ausführung einer Applikation werden neben den eigentlich zu verarbeitenden Daten auch die entsprechenden Instruktionen benötigt, sodass die CPU in den meisten Fällen in jedem Takt einen Speicherzugriff ausführt. **Ein ideales Speichersystem sollte also innerhalb einer Taktperiode die Daten zur Verfügung stellen.** Auf der anderen Seite muss der Hauptspeicher einen möglichst großen Adressraum abdecken, um alle notwendigen Daten bereithalten zu können. Beide Anforderungen stellen

hinsichtlich gegenwärtiger Technologien konkurrierende Kriterien dar. Den besten Kompromiss erzielt man mit der DRAM-Technologie. Bei dieser wird der Schwerpunkt aus Gründen, die später noch erläutert werden, auf die Kapazität gelegt wird.

Diese Fokussierung auf die Kapazität verursacht unter anderem die **Lücke in der Leistungsfähigkeit** zwischen der CPU und dem DRAM-Speicher (siehe Abbildung 1.1 im Einführungskapitel). Die Bezeichnung dieses Sachverhalts als *„memory wall“* bei Wulf [Wulf und McKee, 1995] erweckt den Anschein, dass sich eine Barriere aufbaut, deren Überwindung angestrebt werden sollte. Und in der Tat bildet die gegenwärtige Lösung eine stufenweise Annäherung der Zugriffszeiten durch zusätzliche Module. Man spricht von einer **Speicherhierarchie**, deren Komponenten in Abbildung 2.1 exemplarisch dargestellt sind. Diese Konfiguration ist die meistverwendete Implementierung des Hauptspeichers.

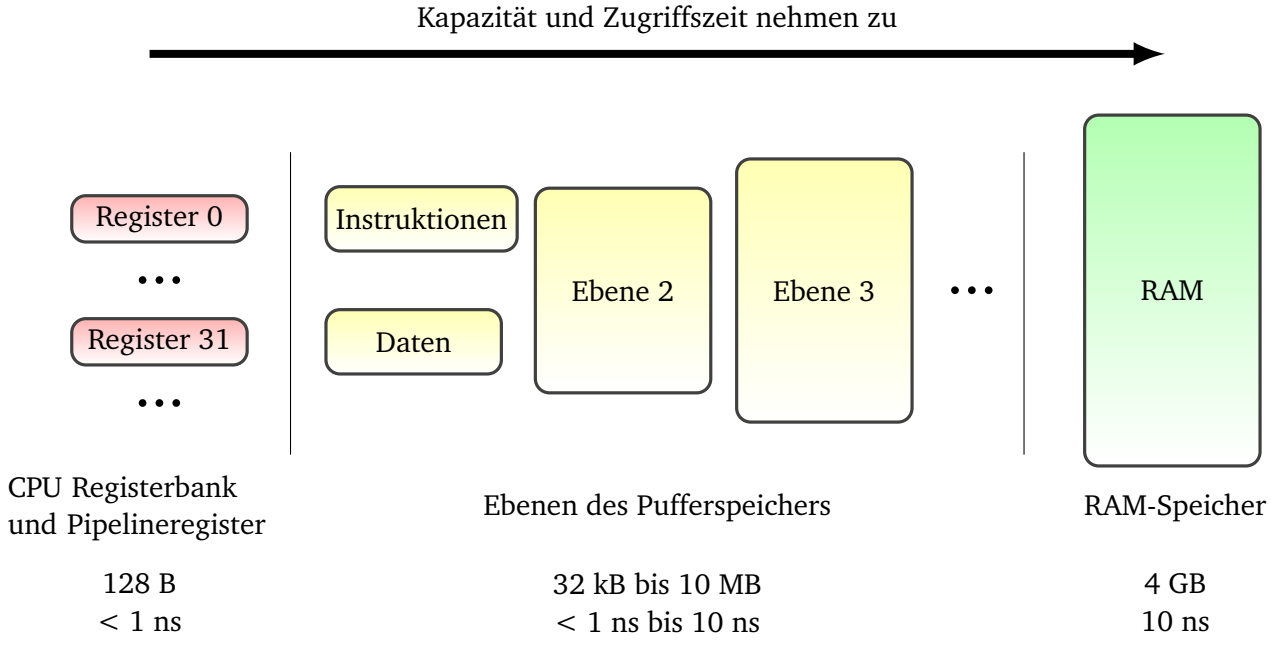


Abbildung 2.1: Speicherhierarchie des Hauptspeichers. Die Angaben zu Kapazitäten und Zugriffszeiten können von System zu System stark variieren.

Die Hauptspeicherhierarchie kann grob in drei Ebenen unterteilt werden: CPU-Register, Pufferspeicher und RAM-Speicher. Diese Einteilung basiert in erster Linie auf der Realisierung der jeweiligen Speicherzellen. An dieser Stelle wird zunächst mit einem RAM-Speicher (***R**andom **A**ccess **M**emory*, Speicher mit wahlfreiem Zugriff) gearbeitet, um anzudeuten, dass die Realisierung dieser - in Bezug auf die Kapazität größten - Komponente nicht vorgegeben ist. Weitere Ebenen der Hierarchie sind allerdings nicht als kapazitätsreduzierende Verkleinerungen zu betrachten. Sie stellen vielmehr eigenständige Komponenten dar. Selbst innerhalb der Unterebenen des Pufferspeichers können deutliche Unterschiede in der Implementierung auftreten [Hennessy und Patterson, 2012]. Daher wird in den folgenden Abschnitten der Pufferspeicher und anschließend der RAM-Speicher detaillierter beschrieben.

Eine gesonderte Stellung in der Speicherhierarchie nehmen die CPU-Register ein. Ihre Zugriffszeit ist in der Regel kleiner als die Taktbreite, sodass ein Zugriff innerhalb eines Taktes erfolgen kann. Manche Register sind nur für die Hardware der CPU „sichtbar“, wie Pipelineregister beispielsweise, während andere zu einer Bank zusammengefasst werden und per Softwarebefehl ansprechbar sind. Die Register werden als Teil der CPU-Logik angesehen, sodass hier eine gewisse Überlappung der Funktionalitäten „Verarbeiten“ und „Speichern“ vorliegt. Dennoch kann an dieser Stelle von einem idealen Speicher auf der untersten Ebene gesprochen werden. Der Pufferspeicher wird zwar in der Regel ebenfalls auf demselben Chip realisiert wie die CPU-Logik, die Einbindung der Register ist allerdings so eng mit der Struktur des Prozessors verbunden, dass eine gesonderte Betrachtung als Speicherkomponente zu kurz greifen

würde. Im Abschnitt 3.1 wird auf die verwendete CPU eingegangen und der Zugriff auf die CPU-Register näher erläutert.

2.1.2 Pufferspeicher (cache)

Der Pufferspeicher (*cache*) behält, wie der Name schon andeutet, die aktuell benötigten Daten und überbrückt die Zugriffszeitdifferenz zwischen der CPU und dem RAM-Speicher soweit, dass auf der untersten Ebene dieses Speichers ein möglichst ideales Verhalten abgebildet werden kann.

SRAM-Zelle

Die schnelle Zugriffsgeschwindigkeit des Pufferspeichers erfordert auch eine physikalische Nähe zur Verarbeitungseinheit. Die eigentliche Speicherzelle für die Daten muss also mit den gleichen Herstellungsprozessen erzeugt werden können, wie die Verarbeitungslogik. Abbildung 2.2 zeigt den schematischen Aufbau einer solchen Zelle, die als SRAM (***S**tatic **R**andom **A**ccess **M**emory*) Zelle bezeichnet wird.

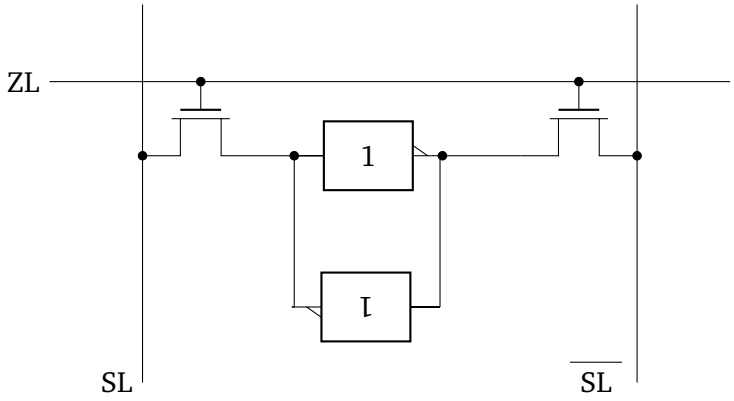


Abbildung 2.2: SRAM-Zelle mit Ansteuerung über Zeilenleitung (ZL) und Spaltenleitung (rechts negiert) [Jacob et al., 2007]

Die Zelle benötigt zwei Transistoren und zwei Inverter, um ein Bit an Daten speichern zu können. Solche Zellen werden als Knotenpunkte in ein Gitter aus Zeilen- und Spaltenleitungen (im negierten und nicht negierten Zustand) platziert. Sie können so über eine entsprechende Spalten- und Zeilenadresse angesprochen werden. Im Kern handelt es sich also ebenfalls um einen RAM-Speicher. Neben dem Herstellungsprozess unterscheidet sich dieser Speicher von den viel größeren (D)RAM durch eine weitere Eigenschaft. Das *static* im Namen bezeichnet den Unterschied zum *dynamic* im DRAM. Die Zelle behält ihre Information, solange die Versorgungsspannung die erforderliche Energie bereitstellt. Es ist also ein flüchtiger Speicher, der während der Ausführung faktisch als nicht-flüchtig betrachtet werden kann[Jacob et al., 2007, Hennessy und Patterson, 2012].

Working Set Model

Die Schnittstelle zwischen der CPU und dem Pufferspeicher benutzt Leitungen innerhalb des Chips, sodass eine hohe Übertragungsgeschwindigkeit möglich ist. Allerdings beanspruchen die SRAM-Zellen durch ihren Aufbau vergleichsweise viel Chipfläche. Das setzt wirtschaftliche Grenzen in Bezug auf die Kapazität eines Pufferspeichers. Außerdem können solche Bitgitter nicht beliebig groß werden, da die Zugriffszeit auf einzelne Bits damit zunehmen würde. Der Pufferspeicher bildet daher immer nur einen kleinen Teil des Hauptspeichers ab. Eine weitere zentrale Eigenschaft der Speichernutzung ermöglicht es dennoch, mit diesem kleinen Ausschnitt die Ausführung einer Applikation zu beschleunigen.

Das Auseinanderlaufen von Zugriffszeiten zwischen der CPU und dem DRAM setzte zwar bereits in den 80er Jahren des letzten Jahrhunderts ein [Jessen, 1996]. Die Kapazität eines schnellen Speichers war allerdings schon vorher in begrenzten Maßen verfügbar, sodass das Konzept eines Pufferspeichers bereits Anfang der 60er Jahre (als „slave-memory“ bei [Wilkes, 1965] beispielsweise) präsentiert wurde. Es stellte sich schon damals die Frage, welche Daten in diesen Pufferspeicher kopiert werden sollen, um die Anzahl der Datenübertragungen auf die langsameren Medien möglichst gering zu halten. Aufgrund von Untersuchungen und Erfahrungen bzgl. der Nutzung eines Speichers stellte sich heraus, dass große Datenpakete die Ausführung eher verlangsamen [Belady, 1966]. Denning formulierte 1970 eine optimale Strategie für die Nutzung der begrenzten Kapazität des Pufferspeichers: das **working set model** [Denning, 1968]. Er fand heraus, dass zufälliges Ersetzen von gepufferten Datenpaketen die höchste Anzahl an Datenübertragungen verursacht, während sein *working set model* die geringste Anzahl aufweist. Diese Eigenschaft ist als **Lokalitätsprinzip** bekannt und umfasst sowohl das räumliche als auch das zeitliche (das auch als räumliche mit einer Distanz zum Nachbarn gleich Null beschrieben werden kann [Gupta et al., 2013]) Zugriffsverhalten.

Die genauen Zahlen für den „working set“ einer Applikation variieren zwar, es lässt sich allerdings folgende Faustformel aufstellen: *„10 % des verwendeten Speichers liefern 90 % aller ausgeführten Instruktionen“*. Die in etwa gleiche Gesetzmäßigkeit tritt auch bei Daten zu [Hennessy und Patterson, 2012]. Das Lokalitätsprinzip ermöglicht es dem Pufferspeicher, die Ausführung einer Applikation auch mit einer geringen Kapazität signifikant zu beschleunigen, erfordert aber zusätzliche Kontroll- und Steuerungslogik, um die häufig genutzten Daten (die 10 %) gesondert zu behandeln.

Die Existenz des Pufferspeichers oder sogar die interne Struktur des Hauptspeichersystems wird bis auf wenige Ausnahmen im Instruktionssatz einer CPU nicht berücksichtigt. Der Zugriff auf den Hauptspeicher ist also transparent. Daher muss die Kontrolllogik des Pufferspeichers selbständig entscheiden, wie die Daten aus dem (D)RAM-Speicher gepuffert werden sollen. Dabei gibt es zwei prinzipielle Ansätze:

- Die Daten können an eine beliebige Stelle im Pufferspeicher platziert werden. In diesem Fall spricht man von einem **voll-assoziativen** Speicher. Bei diesem Konzept muss entschieden werden, welche Daten durch neue ersetzt werden sollen, falls der Speicher schon voll ist. Zudem ist eine Tabelle erforderlich, in der protokolliert wird, von welcher Adresse die Daten kommen, um sie an die richtige Stelle im RAM-Speicher zurückschreiben zu können.

- Der Pufferspeicher wird in Sätze (*set*) aufgeteilt. Die letzten `n` Bits der Adresse entsprechen einer Satzadresse. Der (D)RAM-Speicher wird damit **direkt** im Pufferspeicher **abgebildet**. Die restlichen Bits der Adresse bilden den *tag*, der das jeweilige Datenpaket identifiziert. Bei diesem Konzept ist keine Ersetzungsstrategie notwendig. Sobald ein Datensatz angefordert wird und der neue *tag* mit dem im Pufferspeicher vorhandenen nicht übereinstimmt, wird diese Stelle für neue Daten verwendet.

Beide Konzepte haben Vor- und Nachteile. Darum werden sie in der Regel gleichzeitig im Pufferspeicher realisiert. Wenn die Ersetzungsstrategie von mehreren (`n`) direkt abbildenden Komponenten durch eine übergeordneten Logik gesteuert wird, liegt ein n-assoziativer Pufferspeicher vor. Der Wert des Platzhalters `n` beeinflusst die Größe der Kontrolllogik des Pufferspeichers. Jede direktabbildendende Komponente benötigt einen Vergleicher, um feststellen zu können, ob die angeforderten Daten sich momentan im Pufferspeicher befinden (*cache hit*). Wenn das nicht der Fall ist (*cache miss*), muss eventuell ein Datenpaket ersetzt werden. Die Logik der Entscheidung, wohin die neuen Daten geschrieben werden sollen, wächst ebenfalls mit dem `n`.

Neben der Ersetzungsstrategie kann die Größe eines einzelnen Datenpakets (*cache line*) ebenfalls variiert werden. Sie kann der Datenbreite eines CPU-Registers entsprechen oder ein Vielfaches davon sein. Eine optimale Größe für jede Applikation lässt sich leider nicht bestimmen. Ein kleines Paket ruft häufige Wechsel hervor, wohingegen ein großes Paket längere Zeit zum Schreiben benötigt und die CPU währenddessen entsprechend aufhält [Jacob et al., 2007]. Jedes Datenpaket benötigt einen `tag`, um identifiziert

werden zu können. Gleichzeitig ist dieser `tag` Teil der Auswertungslogik und beansprucht damit Chipfläche, ohne Nutzdaten bereitzustellen. Somit weist ein Pufferspeicher mit einem Vielfachen der CPU-Datenbreite als Datenpaket eine höhere Ausnutzung des Chips in Bezug auf das Verhältnis von Nutz- zu Kontrolldaten auf. Ein zu großes Datenpaket würde die Zugriffsdauer dagegen zu stark erhöhen.

Mithilfe der drei Begriffe (Assoziativität, Datenpaket und Ersetzungsstrategie) lässt sich ein Pufferspeicherzugriff anhand von Abbildung 2.3 demonstrieren. Die untersten Bits der Speicheradresse adressieren innerhalt eines Datenpaketes, wenn das Paket ein Vielfaches der CPU-Datenbreite beinhaltet. Weitere Bits der Speicheradresse wählen den aktuellen Satz aus, der `n`-fach (in Abbildung 2.3 zweifach) aus einer Kombination aus einem `tag`, Datenpaket und Gültigkeitsbit besteht. Die `tags` des Datensatzes entsprechen den obersten Bits der Speicheradresse. Wenn einer der Tags (es sollte maximal nur einer sein) mit dem `tag` der angeforderten Adresse übereinstimmt und wenn der „gültig“-Bit gesetzt ist, können die Daten an die CPU weitergeleitet werden. Anderenfalls meldet der Pufferspeicher einen Fehlzugriff und fordert die Daten von einer höheren Ebene beziehungsweise RAM-Speicher an.

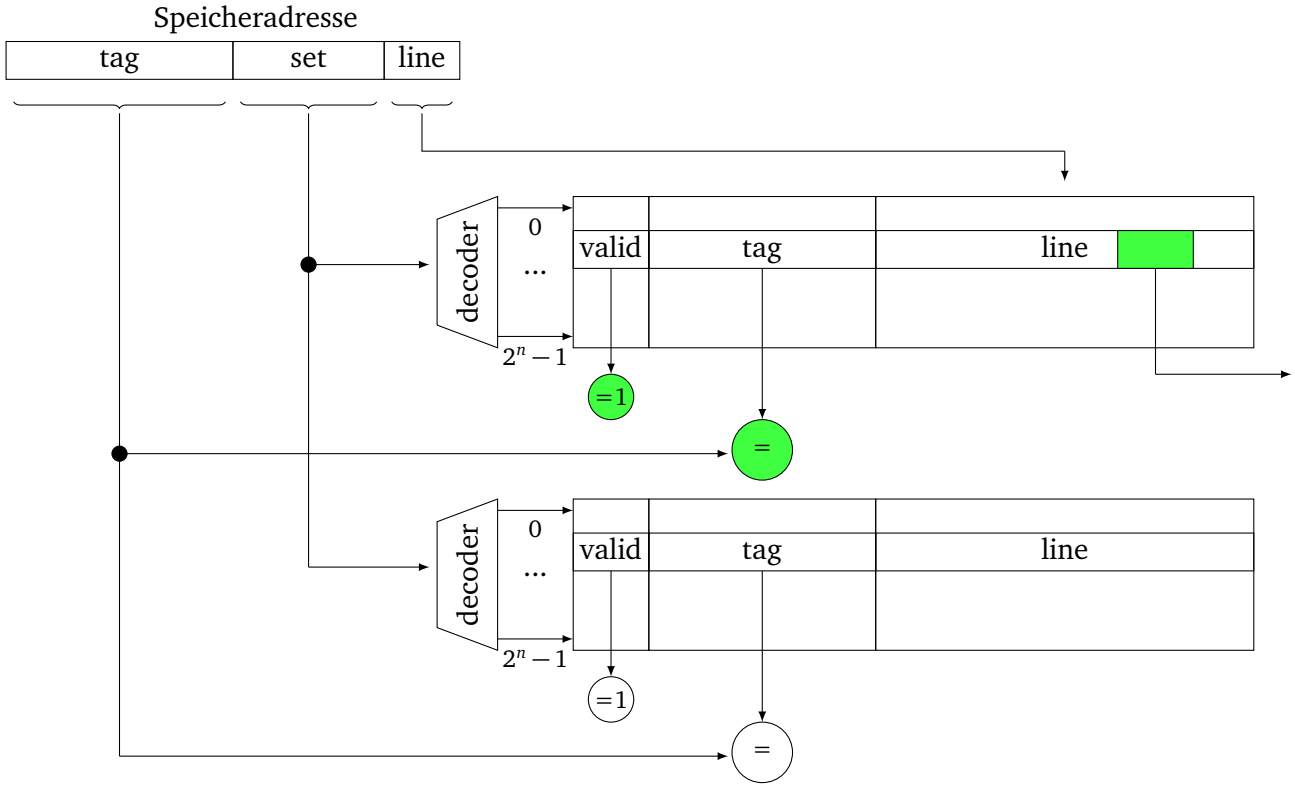


Abbildung 2.3: Interne Struktur eines Pufferspeichers [Jacob et al., 2007]

Schreibzugriff

Aus dem Lokalitätsprinzip kann eine in Abbildung 2.4 dargestellte Verteilung der Adressen nach Zugriffsanzahl gefolgert werden. Der genaue Verlauf der Kurve kann von Applikation zu Applikation variieren.

In Abbildung 2.1 wurde bereits angedeutet, dass sich der Pufferspeicher in mehrere Ebenen unterteilen lässt. Man kann der Zugriffsverteilungskurve entnehmen, dass eine weitere Ebene des Pufferspeichers, die eine größere Kapazität hat, immer noch in der Lage ist, eine große Anzahl an Zugriffen abzudecken. Die Zugriffszeit dieser Ebene ist zwar größer, dennoch ist eine signifikante Beschleunigung durch diese einen Einsatz dieser Ebene feststellbar. Daher teilt sich der Pufferspeicher seit den 90er Jahren des letzten Jahrhunderts in mehrere Ebenen auf [Jessen, 1996]. Auf der untersten Ebene wird in der Regel noch

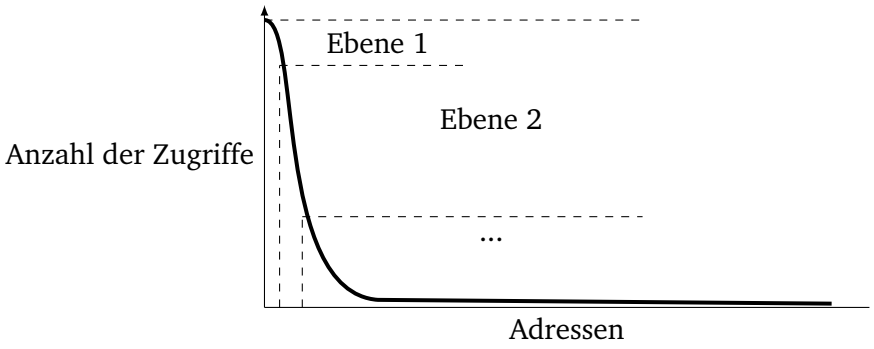


Abbildung 2.4: Darstellung der Verteilung der Adressen, sortiert nach der Anzahl der Zugriffe, und eine mögliche Verteilung auf die Pufferspeicherebenen.

eine weitere Aufteilung in Instruktions- und Datenspeicher vorgenommen. Da die Instruktionen nur gelesen werden, erlaubt es diese Aufteilung, auf eine weitere wichtige Logik des Pufferspeichers im Instruktionsteil zu verzichten: die Verwaltung der Schreibzugriffe der CPU.

Lese- und Schreibzugriffe unterscheiden sich in Bezug auf die Ausführungszeit einer Applikation grundlegend. Ein Prozessor wird in der Regel darauf optimiert, in nahezu jedem Takt eine Instruktion auszuführen. Der Instruktionsspeicher muss also permanent Lesezugriffe bearbeiten. Jede Verzögerung äußert sich unmittelbar in der Ausführungszeit. Diese Randbedingungen gelten auch für den Lesezugriff auf den Datenspeicher. Bei einem Schreibzugriff reicht es dagegen aus, der CPU zu signalisieren, dass die Daten angenommen wurden, auch wenn sie noch gar nicht den SRAM-Speicher erreicht haben. Der Prozessor wartet lediglich auf die Ausführung des Schreibvorgangs, nicht auf die Daten selbst, wie bei einem Lesezugriff. Ein Schreibzugriff stellt eine andere Herausforderung im Pufferspeichersystem dar. Durch die Kombination mit weiteren Ebenen dieses Systems und der erwähnten möglichen verzögerten Ausführung ist es durchaus üblich, dass die Ebenen veraltete Daten beinhalten. Eine Änderung der Daten auf der untersten Ebene wirkt sich erst mit Verzögerung auf die weiteren Ebenen oder gar auf den (D)RAM-Speicher aus.

Diese Dateninkonsistenz stellt für ein System mit nur einem Prozessorkern kein Problem dar, da die Daten zuerst auf der untersten Ebene adressiert werden, wo sie stets aktuell sind. Bei Mehrkernprozessoren können die eben geänderten Daten von einem weiteren Kern gelesen werden, wenn es entsprechende Datenabhängigkeiten gibt. Daher hat die Datenkonsistenz in einem Mehrkernprozessorsystem eine große Bedeutung.

Bei einem Zugriff auf den Pufferspeicher wird als Erstes ausgewertet, ob zu der angelegten Adresse auch Daten im Speicher zu finden sind. Falls ein Fehlzugriff vorliegt, kann bei einem Schreibzugriff geprüft werden, ob die Daten nur weitergeleitet werden (*write not allocate*) oder parallel in den SRAM-Speicher der jeweiligen Ebene geschrieben werden (*write allocate*). Wenn die Daten allerdings vorhanden sind, gilt es, die Datenkonsistenz wiederherzustellen. Dabei gibt es neben einer vollständigen Übertragung aller Daten der jeweiligen Ebene (*cache flush*) zwei Ansätze:

- Die Daten werden in den SRAM-Speicher der jeweiligen Ebene geschrieben und parallel an die obere Ebene weitergeleitet (*write through*). Diese Lösung erfordert einen Adressenpuffer, der die letzten Zugriffe sammelt. Dabei ist es vorteilhaft nur dann ein Datenpaket zu übertragen, wenn sich die aktuelle Pufferspeicheradresse ändert. Anderenfalls speichert der Adressenpuffer mehrfach dieselbe Adresse und läuft viel schneller voll.

- Die Daten werden zunächst nur auf der jeweiligen Ebene geschrieben und nur auf die obere Ebene übertragen, wenn das Datenpaket ersetzt wird(*write back*). Diese Lösung benötigt keinen Puffer, kann aber eine große Dateninkonsistenz verursachen und braucht bei einem Ersetzungsvorgang länger, da die Daten erst zurückgeschrieben werden müssen.

Insgesamt lässt sich zusammenfassen, dass der Pufferspeicher in der Lage ist, das Lokalitätsprinzip sehr effektiv auszunutzen. Die Erkennung des *working set* erfolgt direkt in der Hardware. Das ist auch der Nachteil des Pufferspeichers: eine komplexe Verwaltungslogik. Zudem lässt sich keine allgemein anwendbare Aussage in Bezug auf den optimalen internen Aufbau treffen. Der Aufbau ist stark applikationsabhängig [Hennessy und Patterson, 2012]. Dennoch stellt der Pufferspeicher eine unverzichtbare Komponente in modernen Speichersystemen dar. Sein Markenzeichen ist die Schnelligkeit, während sich ein (D)RAM-Speicher in erster Linie durch eine große Kapazität auszeichnet.

2.1.3 RAM-Speicheraufbau

Ein Speicher mit wahlfreiem Zugriff, ein RAM-Speicher, bildet die Grundlage des Hauptspeichersystems. Die interne Organisation in Bitfelder lässt sich auf allen Ebenen wiederfinden. Der wesentliche Unterschied zwischen dem Pufferspeicher und der Komponente auf der obersten Ebene des Systems (im weiteren Verlauf einfach als **RAM-Speicher** bezeichnet) besteht in der Ansteuerlogik sowie im Aufbau der Speicherzellen. Der RAM-Speicher verfügt über die größte physikalische Kapazität aller Ebenen und bietet dennoch einen schnelleren Zugriff im Vergleich zu üblichen Massenspeichern wie Festplatten, Bandspeichern oder Ähnlichem [Jacob et al., 2007].

Speicherbank

Die Daten eines RAM-Speichers werden in Zellenfelder (*bit array*) gehalten. Die Anzahl dieser Felder bestimmt die Datenbreite einer separat adressierbaren Einheit (*memory bank*). Abbildung 2.5 zeigt den logischen Aufbau einer solchen Einheit.

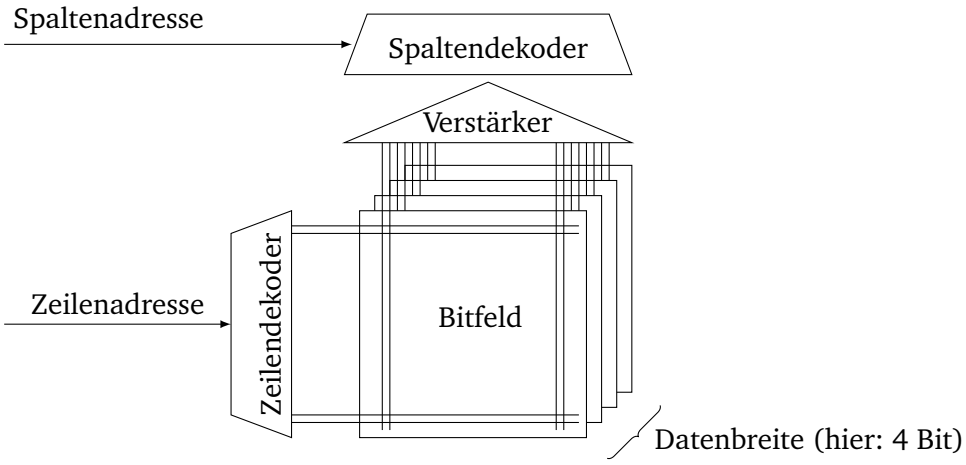
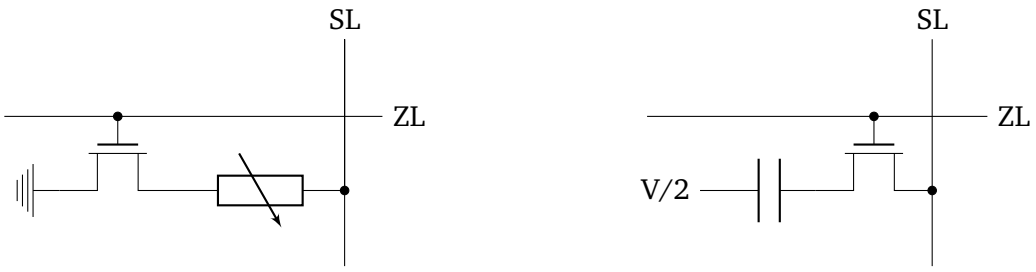


Abbildung 2.5: Logischer Aufbau einer separat adressierbaren Einheit des RAM-Speichers (*memory bank*) [Jacob et al., 2007]

Das Informationsbit braucht - wie ein Punkt in einem kartesischen Koordinatensystem - zwei Adressen, um angesprochen zu werden. Die X-Koordinate ist demnach die Spaltenadresse und die Y-Koordinate die Zeilenadresse. Die Adressen liegen als binärkodierte Zahlen vor. Es darf aber nur eine Zeile beziehungsweise Spalte pro Bitfeld gleichzeitig aktiviert werden. Daher sind zusätzliche Adressdecoder notwendig, die auch entsprechende Treibbausteine beinhalten. Die Bitfelder werden parallel angesteuert und ihre Anzahl bestimmt die Datenbreite, die pro Zugriff über Verstärker gelesen oder geschrieben wird. Ein einzelnes Bitfeld kann somit nicht separat von anderen angesprochen werden. Die Bitfelder werden zusammen zu einer *memory bank* zusammengefasst. Ein RAM-Chip kann über mehrere solcher Bänke verfügen [Keeth et al., 2008].

Der Aufbau einer Speicherzelle des RAM-Speichers kann unterschiedlich realisiert werden. Abbildung 2.6 zeigt zwei mögliche Schaltbilder.



PRAM-Zelle [Zhang und Li, 2009] DRAM-Zelle [Keeth et al., 2008]

Abbildung 2.6: Mögliche Strukturen von Speicherzellen eines RAM-Speichers

Bei der Speicherzelle geht es darum, einen physikalischen Aufbau zu finden, der in der Lage ist, zwei Zustände anzunehmen, den Wechsel kontrolliert zu vollziehen und den aktuellen Zustand wiederzugeben. Und vor allem diesen Zustand über einen gewissen Zeitraum behalten zu können. Wenn dieser Zeitraum groß genug ist (Monate, Jahre), spricht man von einem nichtflüchtigen Speicher. Abbildung 2.6 zeigt auf der linken Seite den Aufbau einer Speicherzelle eines solchen Speichers. Die Informationseinheit wird dabei als Widerstandswert gespeichert (kristalliner oder amorpher Zustand) [Zhang und Li, 2009] und ändert sich nach dem Schreiben nicht. Eine weitere mögliche Speicherquelle stellt die Ausrichtung von Magnetlinien in einem entsprechendem Material dar. Auf dieser Grundlage lässt sich ebenfalls eine nichtflüchtige Speicherzelle realisieren. In diesem Fall spricht man von MRAM [Zhao et al., 2006]. Die interne Struktur des RAM-Speichers lässt Raum für weitere Realisierungsmöglichkeiten der Zellen zu. Es sind andere Randbedingungen, die dazu geführt haben, dass in den meisten modernen Systemen nur eine Speicherzelle zu finden ist: die DRAM-Zelle.

Eine **DRAM-Zelle** (***D**ynamic*) besteht aus einem Kondensator, der über einen Transistor angesprochen werden kann (siehe Abbildung 2.6, rechts). Das Informationsbit wird durch den Ladungszustand des Kondensators bestimmt. Parasitäre Verbindungen führen dazu, dass der Kondensator seine Ladung über die Zeit verliert, sodass er in regelmäßigen Abständen neu beschrieben werden muss. Ein DRAM-Speicher verliert also seine Daten, wenn die Zellen nicht aufgefrischt werden. Daher spricht man vom dynamischen Speicher, im Gegensatz zu statischen Zellen des Pufferspeichers. Zudem muss der DRAM nach einem Neustart oder Reset des Systems seine Daten aus einer nichtflüchtigen Quelle holen. Diese Eigenschaft stellt im Vergleich zu den vorgestellten PRAM- und MRAM-Technologien einen Nachteil dar.

Neben der Flüchtigkeit der Daten spielen Kapazität und Zugriffszeit in modernen Systemen eine wichtige Rolle. Die Kapazität eines RAM-Speichers wird durch seine Packungsdichte bestimmt. Diese Größe wird durch das Verhältnis der Anzahl an Speicherzellen im Bezug auf die benötigte Chipfläche errechnet. Die Verdrahtungslogik und die Ansteuerkomponenten wie Adressdecoder und Verstärker benötigen ebenfalls Chipfläche. Und im Hinblick auf die Packungsdichte kann derzeit - unter Berücksichtigung der Zugriffszeit und wirtschaftlichen Aspekten - keine Technologie bessere Parameter bieten als DRAM [Wu et al., 2009]. Daher wird im weiteren Verlauf ausschließlich DRAM als Realisierungsmöglichkeit des RAM-Speichers betrachtet.

Zugriffsverhalten

Bei einem Zugriff auf eine DRAM-Bank lassen sich mehrere Phasen identifizieren. In Abbildung 2.7 sind die wichtigsten Phasen bei einem Lesezugriff dargestellt.

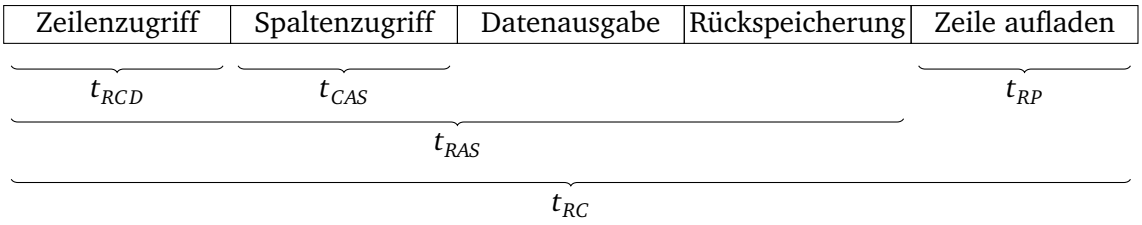


Abbildung 2.7: Verarbeitungsphasen bei einem Lesezugriff auf eine DRAM-Bank [Jacob et al., 2007]

Der erste Vorgang beim Zugriff auf eine DRAM-Bank ist die Aktivierung der adressierten Zeile. Wie jede elektrische Leitung verfügt auch die Zeilenleitung der Bitfelder über parasitäre Kapazitäten und Widerstände, sodass das angeforderte Spannungspotenzial nicht sofort über die gesamte Länge der Zeile erreicht werden kann. Diese Tatsache wirkt sich unter anderem unmittelbar auf die maximale Länge und somit auf die Anzahl der Spalten in einem solchen Bitfeld aus. Nachdem die Zeilenleitung das (in der Regel über der Versorgungsspannung liegende) Spannungsniveau erreicht hat, folgen die Spaltenleitungen, sodass die Zellentransistoren öffnen und ein Ladungsausgleich zwischen der Spaltenleitung und der Zellenkapazität stattfinden kann. Dieser Ladungsausgleich führt zu einer Zu- oder Abnahme des Spannungspotenzials der jeweiligen Spaltenleitung, die wiederum in den Spaltenverstärkern zu zwei stabilen Zuständen führt (die jeweils einer digitalen '1' oder '0' entsprechen).

Die Daten befinden sich somit in den Verstärkern und der Spaltendecoder kann die angeforderte Datenmenge auswählen. Zu diesem Zeitpunkt des Zugriffsvorgangs ist erkennbar, dass die komplette Zeile ausgelesen wurde, auch wenn nur ein einzelnes Bit benötigt wird. Hier spielt die Lokalität (ausführlich beschrieben in 2.1.2) wieder eine Rolle. Es ist sogar vorteilhaft, die benachbarten Daten gleich mitzuleiten. Wenn die Datenbreite der Schnittstelle zwischen CPU- und DRAM-Chip es nicht erlaubt, die komplette Zeile zu übertragen, kann das Datenpaket in mehrere Abschnitte aufgeteilt und in aufeinanderfolgenden Zugriffszyklen (bei doppelter Datenrate, ***D**ouble **D**ata **R**ate: DDR* DRAM ist es auch zu steigender und fallender Taktflanke möglich) an die CPU weitergegeben werden. Man spricht hier von einem Burst-Zugriff.

Weiterhin führt der Aufbau einer DRAM-Zelle dazu, dass das Spannungsniveau zwischen der Kapazität und dem Transistor nach einem Lesezugriff auf das Spaltenleitungsniveau gebracht wurde. Die Dateninformation wurde somit gelöscht und muss neu auf die Zellen übertragen werden. Diese Übertragung beendet den Zugriffsprozess, führt aber dazu, dass während des Zurückschreibens keine weiteren Zugriffe möglich sind. Der (in der Regel weniger zeitkritische) Schreibvorgang verläuft ähnlich wie der Lesevorgang, weist aber den Unterschied auf, dass die Spaltenleitungen beim Spaltenzugriff aktiv auf die entsprechenden Potenziale durch Treibbausteine gebracht werden.

Die Betrachtung des Zugriffsvorgangs auf ein DRAM-Bitfeld zeigt, dass die Datenzugriffszeit davon abhängt, in welchem Zustand sich das Bitfeld während des Zugriffs befindet. Die kürzeste Zeit bietet die Datenübertragung in einem Burst, wenn die Zeile bereits aktiviert ist und die Daten sich in den Leseverstärkern befinden. Am längsten muss man abwarten, wenn die Anfrage in dem Moment kommt, in dem der vorhergehende Prozess gerade mit dem Zurückschreiben beginnt. Diese Dauer kann sich allerdings noch erhöhen.

Das Informationsbit wird in einer DRAM-Zelle mittels Ladung einer Kapazität gespeichert. Diese Ladung lässt sich allerdings wegen parasitärer Verbindungen zu niedrigeren Spannungspotenzialen nicht lange aufrechterhalten. Der Kondensator entlädt sich und seine Ladung muss in regelmäßigen Abständen aufgefrischt werden. Die Auffrischungsperiode hängt in erster Linie von den parasitären Verbindungen

und der Kapazität in Farad ab und beträgt bei modernen Systemen 32 bis 64 ms. Dabei müssen alle Daten eines Bitfeldes intern ausgelesen und wieder neu geschrieben werden. Wenn ein Lesezugriff also genau in dem Moment erfolgt, in dem der Speicher die Auffrischungsphase initialisiert, addiert sich zu der üblichen Zeilen- und Spaltenaktivierungszeit noch die Auffrischungszeit hinzu [Keeth et al., 2008].

Ansätze zur Reduktion der Zugriffszeit

Wenn man den ganzen Zugriffsprozess betrachtet, ergeben sich mehrere Ansätze für eine Verringerung der Zugriffszeit. Als Erstes soll der Informationsaustauschprozess betrachtet werden. Das Informationsbit wird als Ladungszustand eines Kondensators gespeichert. Der Lesevorgang reduziert sich hier physikalisch auf einen Ladungsausgleich zwischen dem Kondensator und der Kapazität der Spaltenleitung. Die beiden Zustände („Speichern“ und „Lesen“) sind in der Abbildung 2.8 exemplarisch dargestellt.

Abbildung 2.8: Zugriff auf einen DRAM-Zellenkondensator und das Schaltbild während des Ladungsausgleichs. Die Richtung des Stromflusses ist hier für U_{Spalte} größer als U_{Zeile} angegeben. Der Sammelwiderstand $R_{Verbindung}$ modelliert vereinfachend die Verbindung zwischen den Kapazitäten.

Vor dem Zugriff ist die Kapazität der Spaltenleitung (C_S) durch den geöffneten Transistor von der Zellenkapazität (C_Z) getrennt. Die Spannungspotenziale betragen jeweils U_S und U_Z. Nachdem der Zellentransistor in den Zustand „geschlossen“ gebracht wurde, findet über einen Sammelwiderstand R ein Ladungsausgleich statt. Unabhängig davon, ob die Zellenspannung größer (eine ’1’) oder kleiner (eine ’0’) als die Spaltenspannung ist, müssen sich die Spannungen in Summe aufheben:

$$U_S + U_R + U_Z = 0 \tag{2.1}$$

Der Strom, der dabei fließt, ändert sich im Laufe der Zeit. Er kann wie folgt angegeben werden:

$$\frac{1}{C_S}\int_0^t i(t)dt + i(t)R + \frac{1}{C_Z}\int_0^t i(t)dt = 0 \tag{2.2}$$

Unter der Annahme, dass zum Zeitpunkt $t = 0$ ein Strom I_0 fließt, lässt sich die Gleichung 2.2 mittels Laplace-Transformation wie folgt in den Frequenzbereich übertragen:

$$\circ\!\!-\!\!\bullet \quad \frac{1}{pC_S}I + IR + \frac{1}{pC_Z}I + \frac{I_0R}{p} = 0 \tag{2.3}$$

$$<=> \quad I\left(\frac{1}{pC_S} + R + \frac{1}{pC_Z}\right) = -\frac{I_0R}{p} \tag{2.4}$$

$$=> \quad I = -I_0\frac{1}{p + \frac{1}{RC_S} + \frac{1}{RC_Z}} \tag{2.5}$$

Diese Gleichung ergibt im Zeitbereich folgende Formel für den Stromfluss.

$$\bullet\!\!-\!\!\circ i(t) = -I_0 \cdot e^{-t\cdot k}, \qquad \text{mit } k = \frac{1}{RC_S} + \frac{1}{RC_Z} \tag{2.6}$$

Der Strom nimmt also exponentiell ab. Für größere k erfolgt die Abnahme noch schneller, wie Abbildung 2.9 zeigt.

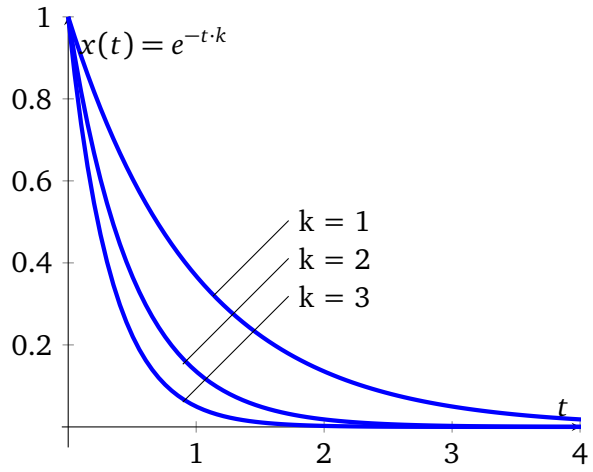


Abbildung 2.9: Verlauf der Eulerfunktion für unterschiedliche Konstanten k

Die Zugriffszeit kann also reduziert werden, wenn k beziehungsweise die Kapazitäten in den Nennern kleiner werden. Die Spaltenkapazität kommt hauptsächlich durch die Länge der Leitung zustande. Ein kleineres Bitfeld würde die Daten also schneller zur Verfügung stellen. Aufgrund der binären Darstellung der Adressen ist nur eine Reduktion um ein Vielfaches der Basiszahl 2 möglich, um eine komplexe Logik für die Adressaufteilung zu vermeiden. Des Weiteren kann auch die Zellenkapazität reduziert werden.

Beide der aufgezeigten Ansätze zur Zugriffszeitminimierung würden die DRAM-Architektur beeinflussen. Eine kürzere Zeilenleitung würde die Größe des Bitfeldes verringern und daher zusätzlichen Verdrahtungsaufwand sowie zusätzliche Adressdecoder erfordern. Insgesamt würde dadurch die Packungsdichte abnehmen. Eine kleinere Zellenkapazität würde schneller die Ladung verlieren, sodass die Auffrischungszyklen kürzer werden müssten. Außerdem würde eine kleinere Zellenkapazität den Ladungsausgleichsvorgang zwar beschleunigen, die Spannungsdifferenz würde allerdings kleiner sein. Diese Differenz müssten dann noch sensiblere Verstärker wahrnehmen. Die Lesefehler könnten zunehmen. Aus technologischer Sicht befindet man sich damit immer noch im Raum der Realisierungsmöglichkeiten und verlagert nur den Schwerpunkt von der Kapazität auf die Zugriffszeit. Und hier spielt ein weiterer Aspekt der DRAM-Welt eine entscheidende Rolle: die wirtschaftliche Betrachtung. Der Preis für DRAM-Speicher orientiert sich in erster Linie an der Kapazität. Schnellere Chips mit geringerer Kapazität können nicht für denselben Preis veräußert werden. Aus diesem Grund blieb die Kernzugriffszeit auf die Bitfelder der DRAM-Speicher bei 10 bis 25 ns über die letzten etwa 20 Jahre unverändert (mündliche Information von Prof. Dr.-Ing. Klaus Hofmann, Darmstadt, 2016).

Der Zugriffsprozess erlaubt allerdings noch einen dritten Ansatzpunkt, um die Gesamtzugriffszeit signifikant zu reduzieren. Nach der Aktivierung der Zeilenleitung und der Aufnahme der Daten in die Leseverstärker können diese Daten direkt an die CPU weitergeleitet werden. Die Übertragung erfolgt nur deswegen in mehreren Schritten, weil die Datenbreite der Schnittstelle zur CPU eine komplette Übertragung in der Regel nicht erlaubt. Der CPU- und der DRAM-Speicherchip stellen zwei separate physikalische Bausteine auf einer Leiterplatte dar. Sie müssen daher über die Pins der jeweiligen Gehäuse und über Leiterplattenleitungen verbunden werden. Die Größe der (verlötfähigen) Pins und die Geometrie der Leiterplattenleitungen stellen technologische Grenzen für die Datenbreite auf. Vielmehr sind es allerdings die Kosten für ein Gehäuse, die diese Anzahl begrenzen. Zudem erfordert die parallele Übertragung über eine (zuvor nicht bekannte) Impedanz der Leiterplattenleitungen zusätzlichen Aufwand für die Anpassung des Abtastzeitpunktes. So hielten Komponenten wie DLL und PLL Einzug in die DRAM-Speicherchips. Neue technologische Lösungen wie die Stapelung mehrerer Chips übereinander

und Herstellung von Querverbindungen zwischen den gestapelten Chips erlaubt es, alle diese Probleme zu umgehen und gleichzeitig die Zugriffszeit zu verringern.

2.2 3D-stacking und 3D-DRAM

Bei einer physikalischen Stapelung (*stacking*) von IC-Chips übereinander, wie es in der Abbildung 2.10 exemplarisch für zwei Chips dargestellt ist, spricht man von einem **3D-IC**.

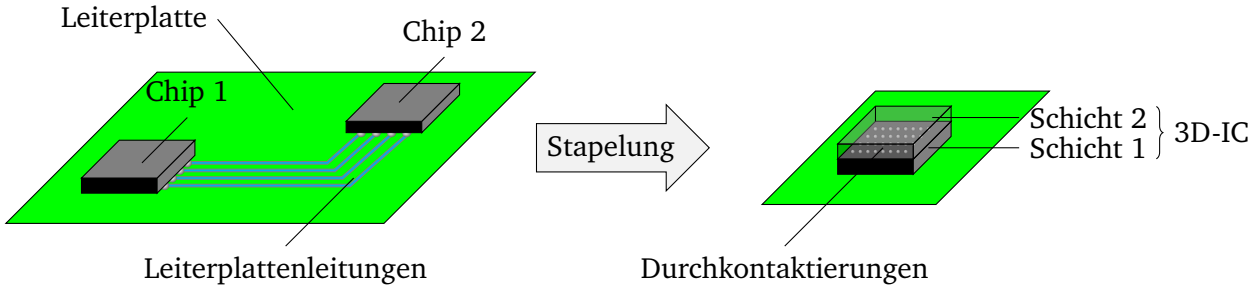


Abbildung 2.10: Stapelung von IC-Chips übereinander

Auf diese Weise können völlig verschiedene Herstellungsprozesse in einem gemeinsamen Chip unterkommen. Auch wenn es streng genommen schon bei der üblichen Prozesstechnik neben Längen und Breiten in die Tiefe geht, bekommt man bei solchen Chips einen klaren Übergang in der Höhe. Daraus ergibt sich die Bezeichnung „drei-dimensionaler IC“. Die Anzahl und die Anordnung der gestapelten Schichten sind nicht vorgegeben. Sie sind unter anderem Gegenstand aktueller Forschungsarbeiten.

2.2.1 Entwicklung der Stapeltechnik

Die wachsende Integrationsdichte der ICs (Mooresches Gesetz [Moore, 1965]) bringt höhere Funktionalität bezogen auf die gleiche Chipfläche. Dabei gab es bereits seit Anfang dieses Prozesses Bestrebungen, heterogene Technologien auf einem Chip zu integrieren.

Ein Chip, der alle nötigen Funktionalitäten in einem Gehäuse vereint, scheint ein erstrebenswertes Ziel zu sein. Die Alternative und damit die aktuelle Lösung für moderne Systeme ist ein Aufbau aus separaten Chips, die auf einer Leiterplatte platziert und über Leitungen auf dieser Platte miteinander verbunden sind. Diese Lösung erlaubt zwar eine Vielzahl an möglichen Kombinationen und Herstellungsprozessen, limitiert aber im Gegenzug die Wahl von Schnittstellen. Jede Verbindung zwischen den Chips muss als Signalleitung an einer der vordefinierten Stellen im Gehäuse, der Lötkugeln (*Pad*), herausgeführt werden, über entsprechende Treibbausteine ein gewisses Spannungspotenzial halten, dann eine (im Voraus nicht immer bekannte) Leiterplattenleitung mit parasitären Komponenten passieren, um dann umgekehrt über eine Lötkugel des anderen Chips aufgenommen zu werden. Dieser Weg lässt sich zwar an vielen Stellen optimieren, verliert aber einen Vergleich mit Verbindungen innerhalb eines Chips in Bezug auf die Betriebsfrequenz, Energieaufnahme, Designfreiheit und Kosten.

Daher wurden bereits in den 70er Jahren des vergangenen Jahrhunderts einzelne ICs in einem gemeinsamen Gehäuse (***M**ulti **C**hip **M**odul*) integriert. Diese Lösung reduzierte lediglich die Anzahl von Bausteinen, die auf einer Leiterplatte platziert werden sollten, eine Optimierung oder gar Stapelung erfolgte allerdings noch nicht. Den nächsten Schritt stellten die *System-on-Chip*-Lösungen dar. Bei diesen Bausteinen werden heterogene Komponenten auf einen gemeinsamen Träger (Substrat) geklebt, gebondet oder in Flip-Chip-Technik verlötet [Lienig et al., 2012]. Die Verbindung untereinander erfolgt im Träger, der nur die für die Außenkommunikation notwendigen Anschlüsse besitzt. Bei dieser Lösung konnten einzelne Chips bereits gestapelt werden, wobei die Verbindung noch über die Chipränder erfolgte (siehe Abbildung 2.11).

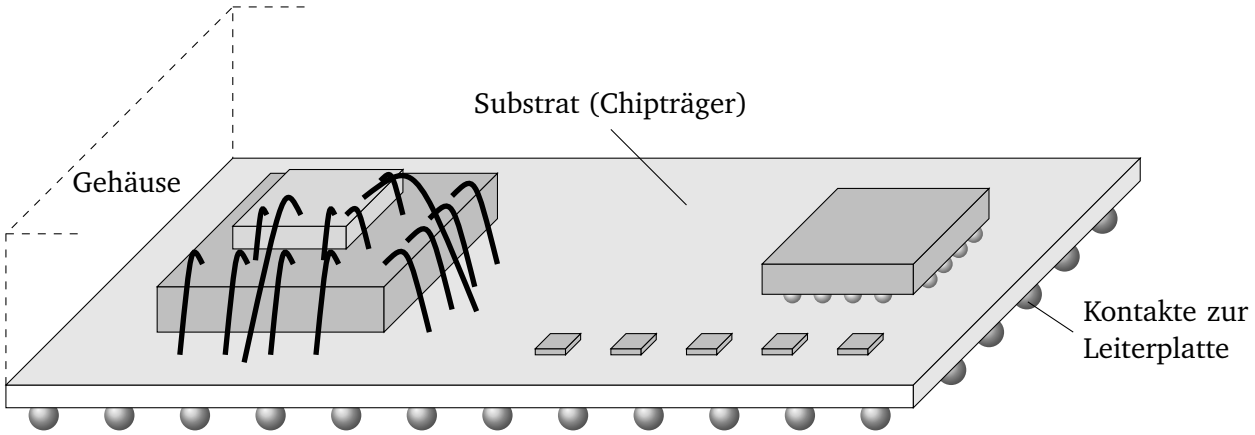


Abbildung 2.11: System-on-Chip [Lienig et al., 2012]

Eine Lösung, bei der man von vollwertigen drei-dimensionalen Schaltkreisen in Bezug auf den Entwurf und die Realisierung sprechen kann, muss das Problem der Querverbindungen lösen. Eine gleichwertige Entwurfsebene - z.B. Verbindungen innerhalb des Chips - konnten die Querverbindungen nur dann erreichen, wenn sie in (theoretisch) beliebiger Zahl an (theoretisch) beliebiger Stelle im Chip platziert werden konnten. Diese Voraussetzungen erfüllen die Durchkontaktierungen der Silizium-Dies (***T**hrough-**S**ilicon-**V**ias*) momentan am besten. Ihre Größe im Vergleich zu anderen Elementen lässt zwar keine beliebige Platzierung zu, im Vergleich zu den Lötkugeln eines Gehäuses stellen sie dennoch einen Quantensprung dar, was Designfreiheit betrifft.

Erst die Einführung dieser Durchkontaktierungen zusammen mit einer präzisen Positionierung einzelner Schichten, die das Schließen der Kontakte erlaubte, eröffneten Entwurfsmöglichkeiten in drei Dimensionen. Den Vorteilen eines 3D-ICs stehen allerdings auch Herausforderungen gegenüber:

- Die gestapelten Siliziumchips tolerieren lokale Temperaturmaxima nur bedingt. Daher muss ihre Entstehung im Entwurfsprozess berücksichtigt werden.
- Zusätzliche Herstellungsprozesse wie Erstellung von Durchkontaktierungen und die Stapelung an sich erhöhen die Herstellungskosten und erzeugen neue Fehlerquellen.
- Sowohl die einzelnen Dies als auch das Gesamtsystem müssen getestet werden, wobei bei Letzteren kein Zugang zu den Durchkontaktierungen besteht.
- Es sind zusätzliche oder angepasste CAD-Werkzeuge für den Entwicklungsprozess notwendig. Dieser Prozess muss entsprechend angepasst werden.

Wie bei jeder neuen Technologie ergeben sich die Vorteile und Herausforderungen im Laufe der Nutzung, sodass der Liste sehr wahrscheinlich noch weitere wichtige Punkte hinzuzufügen sind. Die Stapelung von Siliziumchips kann in vielen Anwendungsbereichen zum Einsatz kommen, beispielsweise mobile Kommunikation, Medizintechnik oder in der Automobilherstellung [Lienig et al., 2012]. In dieser Arbeit geht es primär um den Einsatz von 3D-Technik im Hauptspeichersystem.

2.2.2 3D-DRAM

Die Funktionalität, die die verschiedenen Schichten eines 3D-ICs implementieren, kann sehr unterschiedlich sein. Wenn allerdings ein DRAM-Speicher gestapelt in einem oder mehreren Lagen realisiert wird, spricht man von einem 3D-DRAM.

Organisation eines 3D-DRAM

Die 3D-Technologie erweitert den Entwurfsraum für den DRAM-Speicher. Dabei stellt sich zunächst die Frage, ob die interne Architektur dieses Speichers entsprechend angepasst werden sollte oder bereits vorhandene Entwürfe nur platzsparend übereinander angebracht werden können. Auch die Anzahl und die Kapazität der einzelnen DRAM-Schichten lässt sich nun variieren. Da der DRAM-Speicher Teil des Hauptspeichersystems ist, bietet es sich an, den Übergang zum Pufferspeicher, sei es die Schnittstelle oder die Funktionsaufteilung, neu zu evaluieren. Fragen dieser Art haben mehrere in den letzten Jahren Forschungsaktivitäten ausgelöst. Abbildung 2.12 zeigt ausgewählte Publikationen, eingeordnet nach Hauptansatzpunkten.

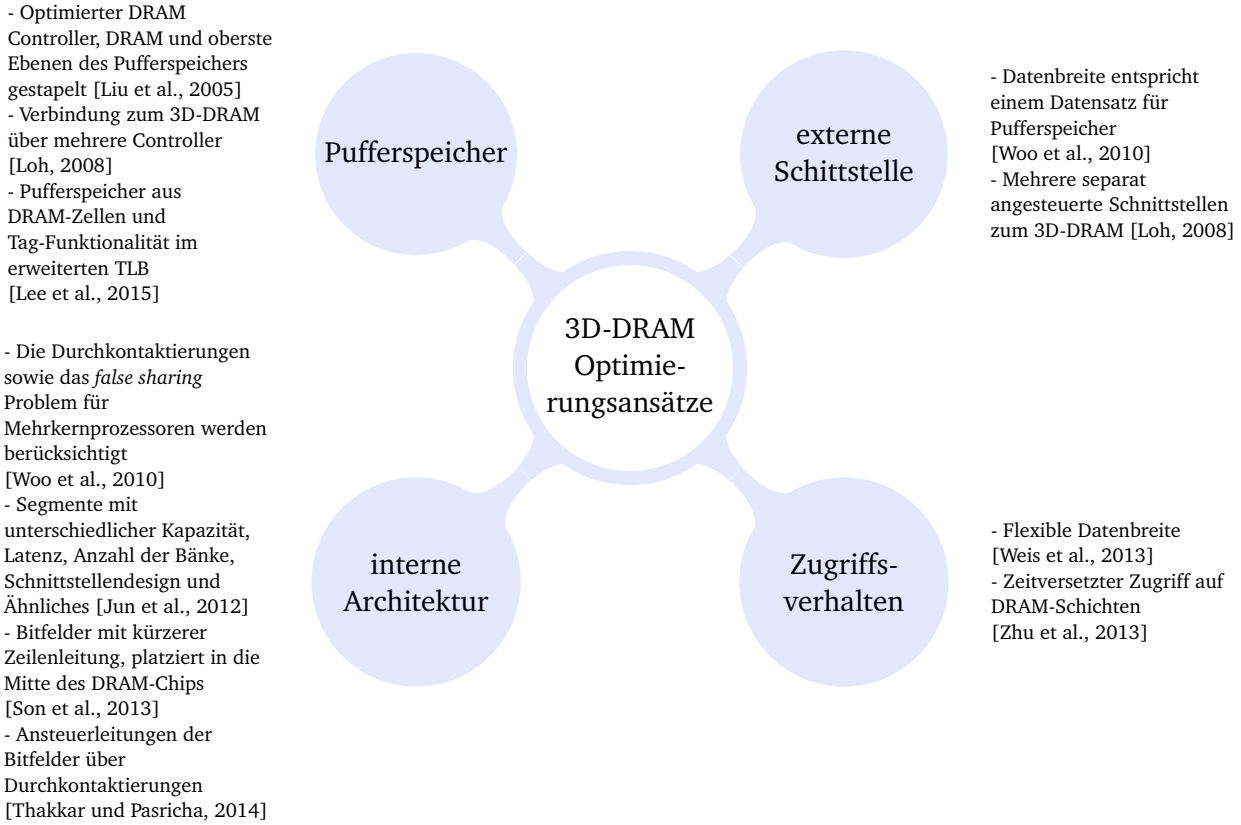


Abbildung 2.12: Optimierungsansätze für 3D-DRAM in den aktuellen Publikationen

Das Hauptspeichersystem besteht neben dem DRAM-Speicher aus unterschiedlichen Ebenen des Pufferspeichers. Das Zusammenspiel zwischen diesen Komponenten bietet einen Ansatzpunkt für Optimierungen der gestapelten Chips. Liu u. a. führten eine Studie mit mehreren variablen Parametern durch [Liu et al., 2005]. Als Vergleichsbasis diente eine übliche Lösung mit zwei separaten Chips: einem mit CPU und Pufferspeicher und einem für DRAM. Diese System verglichen sie unter anderem mit einem gestapelten DRAM-Speicher, der durch einen 50 % schnelleren Controller mit gestapelten Pufferspeicher (Ebene 2, mit und ohne vergrößerte Kapazität) angesprochen wurde. Die Werte wurden mithilfe einer modifizierten Version von SimpleScalar [Austin et al., 2002] gewonnen. Die größte Beschleunigung erreichte das System mit einer gestapelten Ebene 2 des Pufferspeichers, wobei die Kapazität soweit vergrößert wurde, dass alle Speicherzugriffe auf dieser Ebene einen Treffer hatten. Die interne Struktur des DRAM-Speichers wurde dabei nicht verändert.

Lee u. a. schlagen vor, den 3D-DRAM als Pufferspeicher zu benutzen [Lee et al., 2015]. Auf die Speicherung und Auswertung der Tags kann dabei verzichtet werden, denn die nötigen Ersetzungsinformatio-

nen werden bereits im TLB gehalten. Dabei wird der DRAM-Speicher nicht komplett gestapelt realisiert. Es wird nur die große Kapazität durch die DRAM-Zellen für die Pufferspeicherfunktionalität ausgenutzt.

Woo u. a. hatten bei ihren Untersuchungen in erster Linie die Schnittstelle zwischen dem Pufferspeicher und DRAM im Blick [Woo et al., 2010]. Im ersten Schritt variierten sie die Datenbreite des Pufferspeichers (*cache line*) und zählten die Anzahl der Fehlzugriffe des Pufferspeichers während der Ausführung von Vergleichsprogrammen, die zudem nach ihrem Speicherverhalten eingruppiert waren. Dabei stellten sie fest, dass die Zugriffszeit des Pufferspeichers mit der Datenbreite zunimmt und es keinen allgemeingültigen Zusammenhang zwischen den Fehlzugriffszahlen und der Datenbreite gibt. Dennoch bietet die Stapeltechnik die Möglichkeit, den kompletten Datensatz des Pufferspeichers auf einmal zu übertragen. Daher schlugen die Autoren unter anderem mehrere Veränderungen an der DRAM-Architektur vor, die die Durchkontaktierungen und für Mehrkernprozessoren das *false sharing* Problem berücksichtigen. Das veränderte System ist in der Lage, für manche Applikationen bis über zweifache Beschleunigung zu erreichen, wobei der Energieverbrauch gleichzeitig reduziert werden kann.

Die Frage des Energieverbrauchs, charakterisiert durch eine spezielle Größe „Fläche-Energie-Produkt“ (*area-power-product*), ist Gegenstand der Untersuchungen von Jun u. a. [Jun et al., 2012]. Die Autoren stellen fest, dass hauptsächlich der DRAM-Speicher für die Höhe des Energieverbrauchs moderner Serversysteme verantwortlich ist. Daher schlagen sie einen asymmetrischen Aufbau des Speichers vor, bei dem die DRAM-Schicht in Segmente aufgeteilt wird. Unterschieden wird dabei zwischen Kapazität, Anzahl der Bänke, der Bitfelderkonfiguration, dem Schnittstellendesign und Ähnlichem. Für die konkrete Implementierung dieser Segmenten präsentieren Jun u. a. einen Algorithmus, der für eine gegebene Applikation die beste Architektur berechnet. Die übliche Einordnung der DRAM-Speicher nach „Kosten-pro-Bit“ lehnen die Autoren ab. Sie erreichen für ihre Lösung eine Verbesserung des Fläche-Energie-Produkts um bis zu 60 %.

Einen tieferen Einblick in die interne Struktur des DRAM-Speichers gewähren die Vorschläge von Son u. a. [Son et al., 2013]. Sie analysieren die Faktoren, die die Zugriffszeit auf die DRAM-Zellen beeinflussen, und schlagen eine stapelfähige Architektur vor. Dabei soll die Länge der Zeilenleitung reduziert werden, um deren Kapazität zu verringern (siehe Abschnitt 2.1.3). Die kleineren Bitfelder werden dann in der Mitte des Speicherchips platziert. Die sonstigen Peripherieschaltungen werden soweit angepasst, dass Speicherzugriffe in diesen Bereichen schneller verarbeitet werden können. Die Autoren schätzen die Flächenzunahme auf maximal 3 % bei gleichzeitig möglicher Beschleunigung zwischen 20 % und 30 %.

Für einen Einsatz von 3D-DRAM bei Mehrkernprozessoren schlägt Loh eine feinere Aufteilung des DRAM-Speichers in unabhängig voneinander adressierbare Einheiten vor, wobei jede solche Einheit über einen separaten Controller angesteuert werden kann [Loh, 2008]. Die Controller werden geometrisch so platziert, dass sie vertikal eine Verbindung zwischen dem SRAM des Pufferspeichers und den DRAM-Bänken herstellen. Da die oberste Ebene des Pufferspeichers Daten für alle Prozessoren beinhalten kann (*shared cache*), erlaubt die Kontrolllogik des Pufferspeichers auch bei einem Fehlzugriff weitere Datenanfragen (*non-blocking cache*). Dass ein Fehlzugriff vorliegt, wird in entsprechenden Registern festgehalten (*miss status holding register*). Genau diese Register erweisen sich nach Ansicht des Autors als neuer Flaschenhals der Datenübertragung. Als Lösung entwirft er eine neue Datenstruktur, die die Anzahl der Anfragen reduziert. Insgesamt ist der Autor der Ansicht, dass die 3D-Stapeltechnologie noch weiteren Raum dafür bietet, um die Lücke zwischen der Zugriffszeit der CPU und dem DRAM (siehe Abbildung 1.1) zu schließen. Die Möglichkeiten der Stapeltechnik, die vertikale Koordinate in den Designprozess miteinzubeziehen, nutzen Thakkar und Pasricha [Thakkar und Pasricha, 2014]. Sie schlagen vor, die Bitfelder des DRAM-Speichers zu verkleinern und die nötigen Ansteuerleitungen über Durchkontaktierungen zu realisieren. Der Energieverbrauch kann so um über 80 % gesenkt werden, bei gleichzeitiger Beschleunigung um 50 % und bei gleichbleibender Packungsdichte.

Eine umfangreiche Untersuchung in Bezug auf die Architektur des 3D-DRAM führten Weis, Loi u. a. [Weis et al., 2013] durch. Sie benutzten einen synthetisierbaren DRAM-Controller sowie ein von Grund auf implementiertes Speichermodell, um unter Berücksichtigung der Herstellungstechnologie die Or-

ganisation der Speicherbänke sowie der Anzahl der gestapelten Schichten zu evaluieren. Ein Produkt aus Zelleffizienz (Fläche der DRAM-Zellen bezogen auf die gesamte Schichtfläche) und Energieeffizient (Bandbreite bezogen auf Energiebedarf) nimmt für 8 DRAM-Schichten den größten Wert an. Ein optimaler 3D-DRAM besteht nach Untersuchungen der Autoren aus 8 Schichten mit jeweils einer Bank. Darüber hinaus präsentierten die Autoren eine Erweiterung des Zugriffsverhaltens des DRAM-Controllers. Dieser kann im laufenden Betrieb die Datenbreite zwischen 32, 64 und 128 Bit anpassen. Die Logik der Auswahlentscheidung wird im Controller implementiert. Außerdem untersuchten die Autoren den Energieverbrauch des so implementierten 3D-DRAM. Insgesamt erreichten sie eine Reduktion von 83 % pro Bit sowie eine Beschleunigung um bis zu 50 %.

Die Ansteuerung der DRAM-Schichten ist Gegenstand von Untersuchungen von [Zhu et al., 2013]. Die Autoren schlagen einen zeitversetzten Zugriff auf die einzelnen Schichten vor, um so die Bandbreite zu steigern. Der Zugriff wird dabei über eine applikationsspezifische Schicht gesteuert. Außerdem stellen sie die beste Energieeffizienz bei 16 Schichten und 1024 Durchkontaktierungen fest. Sie erreichen ebenfalls 8 Bänke als optimale Lösung, unabhängig von Weis, Loi u. a.

Die vorgestellte Auswahl von Forschungsansätzen zeigt, dass der erweiterte Entwurfsraum, den die Stapeltechnologie bietet, eine Fülle von Ansätzen hervorbringt. Außerdem zeigen diese Arbeiten, dass sowohl in Bezug auf die Leistungsfähigkeit als auch hinsichtlich Energieeffizienz Verbesserungspotenzial vorhanden ist. Eine Verbesserung der Leistungsfähigkeit wird in nahezu allen Arbeiten gezeigt. Der Umfang dieser Verbesserung ist aber unterschiedlich. In dieser Arbeit liegt der Fokus ausschließlich auf der Steigerung der Leistungsfähigkeit durch Stapeltechnik. Dabei wird sowohl das mögliche Potenzial eingegrenzt. Zudem werden die notwendigen Änderungen beschrieben.

Alle gezeigten Arbeiten, diese inbegriffen, zeigen allerdings auch - wie bei jeder neuen Technologie, dass die Evaluation noch eine große Herausforderung darstellt. Auf der einen Seite fehlen speziell angepasste CAD-Werkzeuge, um die vorgeschlagenen Konzepte auch in fertige Testmodule umzusetzen. Auf der anderen Seite wird der DRAM-Speicher auf unterschiedlichen Entwurfsebenen modelliert.

Modellierungskonzepte und Implementierungen

Eine vorangehende Modellierung des 3D-DRAM erlaubt es, Optimierungskonzepte auf ihre prinzipielle Tauglichkeit hin zu prüfen. Dabei stellt jedes Modell nur eine Abstraktion der Realität dar und liefert damit in Abhängigkeit von seinem Umfang nur Näherungen einer möglichen Lösung. Je nach Anforderung kann sich das Modell ausschließlich auf das Verhalten des modellierten Objekts konzentrieren und versuchen, es nachzuahmen. Es kann aber auch versucht werden die interne Struktur des Objekts mit unterschiedlicher Genauigkeit nachzubilden. Ein Verhaltensmodell erlaubt es, systematische Effekte zu untersuchen, während ein Strukturmodell zusätzlich eine mögliche Implementierung bis zu einer bestimmten Stufe vorwegnimmt [Zeigler et al., 2000].

Die Komplexität eines DRAM-Speichers mit seinen unterschiedlichen Komponenten (Bitfelder, Adressdecoder, Verstärker, Datenleitungen und Multiplexern, Verzögerungselementen, Kontrolllogik, Takt- und Energieversorgung und vieles mehr) erschweren die Erstellung eines Verhaltensmodells, insbesondere wenn das Modell mit unterschiedlichen Parametern arbeiten soll. Diese Aussage trifft vor allem auf ein Strukturmodell zu. Dennoch existieren eine Reihe von Implementierungen.

Das häufig in Forschungsarbeiten anzutreffende Modell wurde bei HPLab entwickelt und diente ursprünglich zur Modellierung von Pufferspeichern. Die Modellierung findet innerhalb einer Softwareimplementierung namens CACTI [Wilton und Jouppi, 1996] statt, die neben der möglichen Laufzeit auch die Fläche und den Energieverbrauch schätzt. Ein DRAM wird aus Bitfeldern (im Modell *submat*) zu größeren Felder (*mat*) zusammengeschlossen und mittels H-verteilter Leitungen angesprochen. Somit liegt ein strukturelles Modell vor. 2013 wurde das Modell um die Durchkontaktierung (TSV) erweitert, sodass auch ein 3D-DRAM modelliert werden kann [Chen et al., 2012]. Das Werkzeug CACTI wurde in mehreren Publikationen zur Evaluation von Systemen eingesetzt [Liu et al., 2005, Woo et al., 2010, Jun et al., 2012]. Es lässt sich um weitere spezielle Zwecke erweitern [Li et al., 2011]. Allerdings lässt

sich das Modell nicht separat in ein System einbinden, um beispielsweise die Ausführung einer Applikation zu simulieren. Außerdem hat Weis durch detaillierte Analysen festgestellt, dass das Modell bei Veränderungen der Dimensionsparameter von Speichereinheiten den Flächenwachstum der Peripherieeinheiten linear extrapoliert [Weis et al., 2013]. Dies findet bei realen DRAM-Bausteinen so nicht statt. Wenn die DRAM-Zellen weniger Chipfläche beanspruchen (beispielsweise im Falle kleinerer Bitfelder), dann benötigen sie verhältnismäßig mehr Peripherielogik, um angesteuert zu werden. Diese Eigenschaft von CACTI scheint die Ergebnisse allerdings nicht zu verfälschen, da die Untersuchungen von Weis u. a. ohne CACTI und die Untersuchungen von Zhu u. a. mit CACTI [Zhu et al., 2013] gleiche Ergebnisse in Bezug auf die Organisation des 3D-DRAM liefern (siehe Abschnitt 2.2.2).

Eine ähnliche Lösung bietet DRAMSim von Bruce Jacob an [Jacob, 2006]. Ein Unterschied besteht darin, dass der Fokus bei diesem Modell auf dem Verhalten liegt. Mit diesem Werkzeug lassen sich Laufzeiten von Programmausführungen simulieren. Es lässt sich in ein System einbinden, dass eine Schnittstelle zur Programmiersprache C++ bietet. Das Modell greift auf Daten von vorhandenen DRAM-Chips des Speicherherstellers Micron zu, lässt sich aber auch um andere Daten erweitern. Die dreidimensionale Stapelung von DRAM spielte zur Zeit der Erstellung der Software noch keine Rolle und das Werkzeug wurde seit 2006 nicht mehr aktualisiert. Dennoch bietet das Modell mehrere Ansatzpunkte dafür, die Modellierungskonzepte auf einen 3D-DRAM zu übertragen.

Ein weiteres Softwarewerkzeug, [DRAMPower, 2012], konzentriert sich auf die Schätzung des Energie- und Leistungsverbrauchs eines DRAM-Bausteins. Dabei wird ebenfalls auf Daten von Lösungen des Herstellers Micron zugegriffen, wobei die Datenliste im Vergleich zum DRAMSim aktueller ist. DRAMPower berücksichtigt die Strukturen eines DRAM-Bausteins, um in erster Linie über das Zugriffsverhalten auf den Energieverbrauch zu schließen. Die Zugriffsdaten werden über eine externe Datei importiert. Für eine Einbindung in ein vorhandenes System eignet sich das Modell daher nur bedingt.

Ganz anders verhält es sich mit dem ebenfalls sehr weit verbreiteten Modell des Herstellers Micron [Micron Technology, Inc., 2006]. Seine in Verilog implementierten Modelle des DDR2- und DDR3-Speichers können in eine weitere Umgebung integriert werden und das Systemverhalten nachbilden. Das Modell stellt in erster Linie ein Verhaltensmodel dar und kann mit Daten von existierenden DRAM-Chips sein Verhalten entsprechend anpassen. Die interne Struktur des DRAM-Speichers wird über Aufrufe von entsprechenden Verilog-Funktionen nachgebildet, sodass das Modell dazu in der Lage ist, das Verhalten detailliert nachzubilden.

2.3 Laufzeitmessung

Die Analyse der Performanz eines Systems umfasst ein breites Forschungsgebiet. Ein Versuch, auf einzelne Aspekte der Leistungserfassung einzugehen, würde den Rahmen dieser Arbeit sprengen. Daher beschäftigen sich die folgende Abschnitte speziell mit den für diese Arbeit relevanten Aspekten. Die wichtigste Kenngröße bei den durchgeführten Untersuchungen war die Laufzeit der Ausführung einer Applikation. Im nächsten Abschnitt wird diese Größe definiert. Im darauf folgenden Abschnitt wird das Messsystem eingegrenzt.

2.3.1 Zeitmessung in einem Teilsystem

Laufzeit bzw. Zeit lässt sich als Differenz zwischen zwei Zeitpunkten angeben, wie in Abbildung 2.13 exemplarisch dargestellt ist.

Für ein System, das eine Applikation ausführt, stellen sich jedoch die Fragen nach dem Beginn und Ende der Ausführung komplexer dar, als es auf den ersten Blick erscheint. Für eine Vergleichbarkeit reicht es zunächst aus, die Messbedingungen konstant zu halten und nur die untersuchte Variable zu verändern. Wenn diese notwendige Bedingung erfüllt ist, stellt sich die Frage des Realitätsbezugs der Messbedingungen. Die Vergleichbarkeit erfordert gleiche Vorbedingungen und grenzt damit die Anzahl der möglichen Betriebszustände ein. Schon deswegen kann ein Messverfahren nur eine Annäherung an

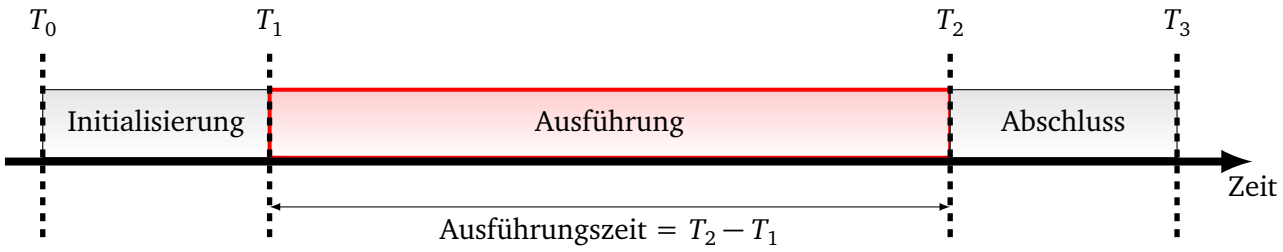


Abbildung 2.13: Laufzeitmessung von einer Applikationsausführung, angepasst von [Ferrari, 1978]

die Realität sein [Jain, 1991]. Neben der Vergleichbarkeit möchte man aber ein möglichst realistisches Messszenario haben.

Weiterhin muss das System definiert sein, auf dem die Messung ausgeführt wird. Wenn man beispielsweise nur die Schnittstelle zwischen einem DRAM-Speicher und der CPU betrachtet und die Betriebsfrequenz erhöht, ergibt sich eine höhere maximale Durchsatzrate. Daraus kann aber nicht gefolgert werden, dass auch die Gesamtausführung entsprechend schneller wird. Diese Schnittstelle stellt nur einen Teil eines größeren Systems dar. Wenn allerdings nur ein Teil des Systems beschleunigt wird, hängt die Gesamtbeschleunigung vom Anteil dieses Teilsystems am gesamten System ab, wie es das Gesetz von Amdahl formuliert [Amdahl, 1967]:

$$s = \frac{1}{(1-p) + \frac{p}{s_p}} \qquad \text{mit}$$

$s =$ Gesamtbeschleunigung
$p =$ Anteil am Gesamtsystem
$s_p =$ Beschleunigung des Anteils

Die Originalpublikation von Amdahl beschäftigte sich zwar mit Beschleunigung durch parallele Ausführung, die Formel lässt sich jedoch auch für andere Szenarien anwenden. Wenn ein Teil des Systems beschleunigt ausgeführt wird, wirkt sich diese Beschleunigung um so mehr auf die Gesamtausführung aus, je größer der Anteil dieser Komponenten an der Gesamtausführung ist. Daher ist jegliche Optimierung des DRAM-Speichers nur dann sinnvoll, wenn sein Anteil an der Gesamtausführung auch entsprechend signifikant ist. Wie groß dieser Anteil ist, lässt sich leider nicht im Voraus für jede Applikation berechnen, auch wenn das Lokalitätsprinzip 2.1.2 eine gewisse Abschätzung zulässt. Doch bevor von einem DRAM-Anteil gesprochen werden kann, muss das Gesamtsystem definiert werden.

2.3.2 Laufzeitmessung mit DRAM-Speicher

Um die die Zeit zu messen, die für die Ausführung einer Applikation gebraucht wird, benötigt man neben der CPU, dem Speicher und der Applikation auch die Eingangsdaten [Jain, 1991], wie Abbildung 2.14 zeigt.

Zunächst ist die Frage zu beantworten, wann die Messung beginnen soll. Am naheliegendsten scheint der Zeitpunkt zu sein, an dem die erste Instruktion der Applikation angefordert wird. Das setzt allerdings voraus, dass der Speicher alle Instruktionen bereits enthält. Die Eingangsdaten werden in der Regel nach einer Initialisierungsphase, die abhängig von der Applikation auch die meiste Zeit beanspruchen kann, angefordert. Um eine Vergleichbarkeit herzustellen, sollten diese Daten ebenfalls bereits im Speicher vorhanden sein oder mit einer immer gleichen Verzögerung - beispielsweise mithilfe eines Zusatzmoduls - in den Speicher übertragen werden. Der Speicher darf keine weiteren Daten enthalten, um eventuelle Nebeneffekte für die Vergleichbarkeit auszuschließen.

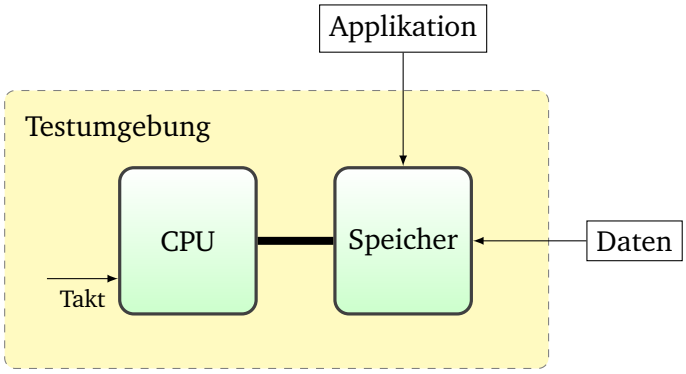


Abbildung 2.14: Komponenten des Messsystems zur Laufzeiterfassung einer Applikation

Neben der Tatsache, dass reale DRAM-Bausteine nach dem Einschalten erst mit Daten beschrieben werden müssen, durchlaufen sie eine Initialisierungsphase, in der der Kommunikationskanal zwischen dem Speicherchip und dem Zugriffschip analysiert wird, um eine möglichst fehlerfreie Übertragung zu gewährleisten. Es ist anzunehmen, dass diese Phase für einen 3D-DRAM weniger umfangreich ausfallen kann, da der Übertragungskanal im Voraus viel besser bekannt ist. Es stellt sich also die Frage, ob diese Initialisierungsphase in die Messzeit einzubeziehen ist oder nicht.

Der Schlusszeitpunkt der Ausführung ist ebenfalls nicht eindeutig bestimmt. Die verarbeiteten Daten werden während der Ausführung in den Speicher zurückgeschrieben. Diese Daten erlauben gleichzeitig eine Aussage darüber, ob die Ausführung auch fehlerfrei verlaufen ist. Da ein reales Speichersystem aber über unterschiedliche Ebenen verfügt, wie im Abschnitt 2.1 beschrieben wurde, besteht für eine gewisse Zeit eine Dateninkonsistenz zwischen diesen Ebenen. Eine vollständige Rückspeicherung des Pufferspeichers aktualisiert alle Daten, benötigt allerdings Zeit, die für die Ausführung einer Applikation auf den ersten Blick nicht notwendig ist. Wenn man aber Aussagen über den DRAM-Speicher treffen möchte und die Rückspeicherung eine Vielzahl an Zugriffen erzeugen kann, muss auch über die Behandlung dieser Phase für die Messung entschieden werden.

Der nächste Punkt betrifft die Applikation selbst. Die meisten Applikationen stellen eine Vielzahl an Konfigurationsmöglichkeiten bezüglich der Verarbeitung der Daten über entsprechende Parameterwerte dar. Diese Werte beeinflussen aber schon prinzipiell die Ausführung. Für die Vergleichbarkeit reicht es aus, diese Werte für alle Messungen konstant zu halten. Sie müssen aber dennoch festgelegt werden. Weiterhin ist es vorteilhaft, mehrere Applikationen zu nutzen. Hierbei stellt sich die Frage, welche und wie viele Applikationen getestet werden sollen. Die Applikationen haben den Vorteil, einen realen Betrieb am besten abzubilden. Ob ihre Menge allerdings die Fragen hinreichend beantworten kann, bleibt an dieser Stelle offen.

Eine Applikation benötigt nocht Daten, die sie bearbeiten soll. Neben der Stützung der Untersuchung auf mehrere Applikation können zusätzlich der Inhalt und die Größe der Daten für die Messung variiert werden. Die Ausführungszeit der gleichen Applikation kann sich signifikant unterscheiden, wenn die Daten unterschiedliche Inhalte haben. Außerdem kann dieser Inhalt und/oder die Größe einen Einfluss auf die Konfigurationsparameter der Applikationen haben. Insbesondere die Größe der Eingangsdaten spielt für ein Speichersystem mit Pufferspeicher und DRAM-Speicher eine große Rolle. Der Pufferspeicher ist in seiner Größe limitiert und kann sich damit unterschiedlich auf das Gesamtsystem auswirken, je nach dem, ob die Daten eine bestimmte Größe überschreiben.

Der letzte Punkt betrifft die Ausführung an sich. Auch ein System mit idealem Speicher benötigt Zeit für die Ausführung. Diese Zeit - wie die Zeit, in der der Pufferspeicher die CPU anhält, um Daten zwischen den Ebenen zu übertragen - wird auch dann benötigt, wenn ein DRAM-Baustein in gestapelter Form zur Verfügung steht. Die ideale Laufzeit lässt sich mit einem Modell eines idealen Speichers messen. Die Zeit, die der Pufferspeicher für sich beansprucht, lässt sich allerdings nur bedingt ermitteln. Die einzelnen Ebenen des Pufferspeichers stellen eigenständige Komponenten dar, die über eine eigenständi-

ge Kontrolllogik verfügen. Diese Logik kann Aktualisierungsroutinen auslösen, während DRAM-Zugriffe ausgeführt werden. Damit wird die Wartezeit doppelt genutzt. Diese Situationen lassen sich nur schwer zeitlich lokalisieren. Allerdings ist ihre Wahrscheinlichkeit gering, sodass ihr Einfluss auf die Gesamtlaufzeit eher als gering einzuschätzen ist. Tendenziell wird so der DRAM-Anteil an der Gesamtlaufzeit aber geringfügig unterschätzt. Zuletzt sei noch der DRAM-Controller erwähnt. Er verwaltet die DRAM-Speicherzugriffe. Seine Bestandteile können sich beim Übergang zum 3D-DRAM ändern.

Alle diese Punkte zeigen, dass man nie von *der* Laufzeit einer Applikation sprechen kann. Es ist vielmehr möglich, Zusammenhänge in einer Messumgebung, die der Realität möglichst nahekommt, festzustellen.

2.4 Hypothesen der Arbeit

2.4.1 Aufstellung der Hypothesen

Die Nutzung des Speichers lässt also Gesetzmäßigkeiten erkennen, die als Lokalitätsprinzip beschrieben werden. Dieses Prinzip besagt, dass die Anzahl der Zugriffe keineswegs zufällig auf die Adressen verteilt ist. Manche Funktionen und Datenstrukturen werden besonders häufig angesprochen. Diese Bestandteile umfassen insgesamt ungefähr 10 % der genutzten Adressen, sodass folgende Verteilung der Adressen (sortiert nach Anzahl der Zugriffe) angenommen werden kann:

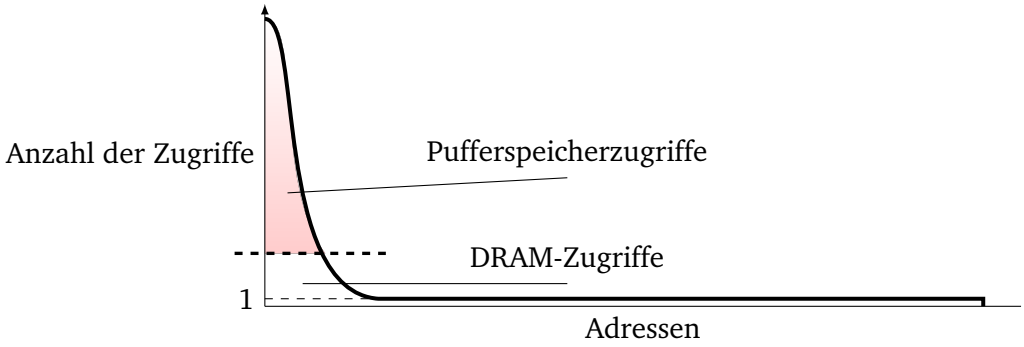


Abbildung 2.15: Sortierte Verteilung der Anzahl der Speicherzugriffe und der Anteil der gepufferten Zugriffe (oberhalb der gestrichelten Linie)

Bevor die Daten gepuffert werden können, müssen sie vom DRAM gelesen werden. Daher kann der Pufferspeicher niemals die ’1’-Grenze nach unten überschreiten. Es werden zwar nur die Basisadressen von Datenpaketen direkt beim DRAM angefragt, durch eine Burst-Übertragung müssen aber alle weiteren Adressen innerhalb des Bursts zumindest intern ausgewählt werden. Diese Adressen müssen zwar nicht zwangsläufig mit den entsprechenden CPU-Adressen übereinstimmen, dennoch kann zusammenfassend angenommen werden, dass auf jedes benutzte und adressierbare Datenwort mindestens einmal zugegriffen wurde. Daraus folgen zwei mögliche Szenarien:

Nach dem ersten Zugriff befinden sich alle Daten im Pufferspeicher, sodass keine weiteren DRAM-Zugriffe mehr auf die jeweiligen Adressen stattfinden. Die Grenze für die Pufferspeicherzugriffe stimmt mit der ’1’-Grenze überein. Dieses Szenario stellt zwar einen Grenzfall dar, ist aber für eine entsprechende Pufferspeichergröße durchaus als realistisch einzustufen.

Beim zweiten Szenario wird angenommen, dass eine signifikante Menge der Adressen mehr als einmal vom DRAM-Speicher angefragt wird. Ausgehend von diesen beiden Szenarien lassen sich folgende Hypothesen formulieren:

Hypothese 1

Wenn alle weiteren Zugriffe auf die Adressen, nach dem ersten Lesen im DRAM, gepuffert sind, kann nur eine Zugriffszeitreduktion des DRAM-Speichers die Ausführungszeit der Applikation verringern.

Hypothese 2

Je größer die Menge an Adressen ist, die mehr als einmal im DRAM angefragt werden, desto mehr kann die Laufzeit durch Pufferspeicherfunktionalität im DRAM verringert werden.

Pufferspeicherfunktionalität bedeutet, dass häufig genutzte Daten mit einer geringeren Latenz zur Verfügung gestellt werden. Wenn allerdings die gesamten Daten bereits im eigentlichen Pufferspeichers gehalten werden, dann bringt diese Funktionalität im DRAM-Speicher keine weiteren Vorteile mehr. An dieser Stelle lässt sich bereits eine Abhängigkeit von der Größe der zu verarbeitenden Daten erkennen. Wenn diese Menge groß genug ist (was das bedeutet, muss ebenfalls eingegrenzt werden), postuliert die Hypothese 2 die Möglichkeit einer weiteren Verbesserung. An dieser Stelle kann dieser Gedanke zu einer weiteren, zugegeben gewagten, These führen:

Hypothese 3

Außer der Implementierung der Pufferspeicherfunktionalität kann kein weiterer Ansatz höhere Leistungssteigerung bewirken.

Die Überprüfung dieser Hypothese setzt allerdings voraus, dass alle weiteren Ansätze bekannt sind und getestet werden können. Es ist aber nicht nur im Rahmen dieser Arbeit unmöglich, alle bekannten Ansätze durchzugehen. Daher wird hier nicht der Anspruch erhoben, die aufgestellte Hypothese vollständig bestätigen zu können.

2.4.2 Überprüfungsansatz

Die wissenschaftliche Evaluation einer Fragestellung kann auf unterschiedliche Weisen erfolgen. Bei den Ingenieurwissenschaften lassen sich drei mögliche Ansätze benennen:

- Mathematische Herleitung
- Simulation des Systems
- Experimentelle Implementierung und Ausführung

Im Idealfall kann ein Effekt, der mathematisch hergeleitet wurde, simulativ und anschließend auf einer realen Hardware bestätigt werden. Der erste Schritt benötigt ein theoretisches Modell des Systems, der zweite ein entsprechend detailliertes Simulationsmodell und der dritte eine existierende Hardware. Die mathematische Grundlage für diese Arbeit bietet das *working set model* von Denning. Die experimentelle Überprüfung auf einer realen Hardware muss allerdings außerhalb des Rahmens dieser Arbeit bleiben, da die Realisierung der angedachten Architektur des DRAM-Speichers aus zeitlichen und wirtschaftlichen Gründen nicht möglich war.

Diese Einschränkung gilt allerdings nicht für andere Komponenten der Testumgebung. Insbesondere für den Prozessor und den Pufferspeicher wurden die Implementierungen auf einem FPGA-Board

getestet. Dieser Ansatz wurde unter anderem gewählt, um einen möglichen nächsten Schritt zur Hardwarerealisierung - soweit möglich - zu verkürzen. In Bezug auf die Hypothesen erfolgte die Überprüfung simulativ. Der Aufbau der Testumgebung umfasste folgende Schritte:

- Modellierung der Messkomponenten wie CPU, Pufferspeicher und DRAM-Referenzmodell.
- Modellierung des 3D-DRAM-Modells.
- Portierung der Software und Auswahl der Eingangsdaten.
- Messung der Laufzeit und Vergleich vom 3D-DRAM und Referenzmodell.

Die folgenden zwei Kapitel beschreiben diese Schritte. Danach werden die Ergebnisse präsentiert.

3 Modellierung der Testkomponenten

3.1 CPU

Die Wahl einer Ausführungseinheit, hier die CPU, scheint für eine Untersuchung, die in erster Linie Speicher betrifft, eher zweitrangig zu sein. Tatsächlich beeinflusst die Wahl einer solchen Ausführungseinheit maßgeblich die Ergebnisse der Speichernutzungsanalyse. Eine Speicherschnittstelle mit einer großen Bandbreite nutzt voraussichtlich in erster Linie Grafikprozessoren (GPUs) oder Prozessoren, die mit mehreren Daten gleichzeitig arbeiten (***S**ingle **I**nstruction **M**any **D**ata*). Eine Ausführung auf einem Mikrocontroller, der pro Takt beispielsweise maximal 8 Bit an Daten benötigt, wird eher von geringer Latenz statt großer Bandbreite profitieren. Letzten Endes kommt es aber auch auf die Applikation oder vielmehr auf die auszuführenden Instruktionen, also auch auf den Compiler und die benutzten Bibliotheken, an. Folgendes lässt sich bei Untersuchungen immer wieder beobachten: „Auch wenn eine Lösung keine signifikanten Verbesserungen im Allgemeinen erzielen kann, lässt sich sehr oft eine spezielle Anordnung finden, die genau von dieser Lösung profitiert."

Bei dieser Arbeit soll dagegen eine andere Frage beantwortet werden: „Ist ein 3D-DRAM **grundsätzlich** in der Lage, ein System signifikant zu beschleunigen?" Diese Fragestellung erfordert eine CPU, die für keine speziellen Aufgaben konzipiert ist. Damit ist das erste grundlegende Kriterium für die Wahl formuliert. Des Weiteren würde eine quelloffene Implementierung eventuelle Änderungen für den Testaufbau ermöglichen. Nicht zuletzt sollte ein Compiler zur Verfügung stehen, um vorhandene Applikationen portieren zu können. Eine Prozessorstruktur, die für keine speziellen Aufgaben konzipiert wurde, stellt die RISC-Architektur dar. Diese Architektur ist schon per definitionem allgemein gehalten. Ein Vertreter dieser Architektur ist die MIPS-Spezifikation, deren quelloffene Implementierung durch einen CPU-Kern namens *plasma* [Rhoads, 2013] realisiert wurde. Die Spezifikation wird durch den GNU-Compiler unterstützt, sodass alle notwendigen Kriterien für die Wahl der CPU mit diesem Kern erfüllt sind.

3.1.1 CPU-Kern namens *plasma*

John L. Hennessy entwickelte Anfang der 80er Jahre zusammen mit seinen Mitarbeitern an der Stanford-Universität einen RISC-Befehlssatz [Hennessy und Patterson, 2012]. Die dazugehörige Architektur wurde MIPS (***M**icroprocessor without **I**nterlocked **P**ipeline **S**tages*) genannt und diente in erster Linie didaktischen Zwecken. Der MIPS-Befehlssatz wurde über die Jahre weiterentwickelt und kommerziell lizenziert. Inzwischen bietet der Befehlssatz Versionen sowohl für energieverbrauchskritische Mikrocontroller als auch für Mehrkernsysteme. Um eventuellen lizenz- oder patentrechtliche Fragen aus dem Weg zu gehen, bot sich die Version I aus dem Jahr 1985 an, da sie auf der einen Seite keine speziellen Befehle definiert und auf der anderen Seite alle relevanten Assemblerbefehlsbereiche abdeckt.

Der *plasma*-Kern implementiert bis auf wenige Ausnahmen genau diese erste Version des MIPS-Befehlssatzes. Der Quellcode wurde von Steve Rhoads in der Hardwarebeschreibungssprache VHDL verfasst und ist unter [Rhoads, 2013] frei verfügbar. Die Implementierung wurde bewusst bis auf ein Minimum reduziert, sodass eine Einarbeitung mit vergleichsweise geringem Aufwand möglich ist. Außerdem kann der frei verfügbare GNU-Compiler eingesetzt werden, um einen Maschinencode für diesen Kern zu erzeugen.

Die externe Schnittstelle des *plasmas* verfügt nur über wenige Signale, wie Abbildung 3.1 zeigt.

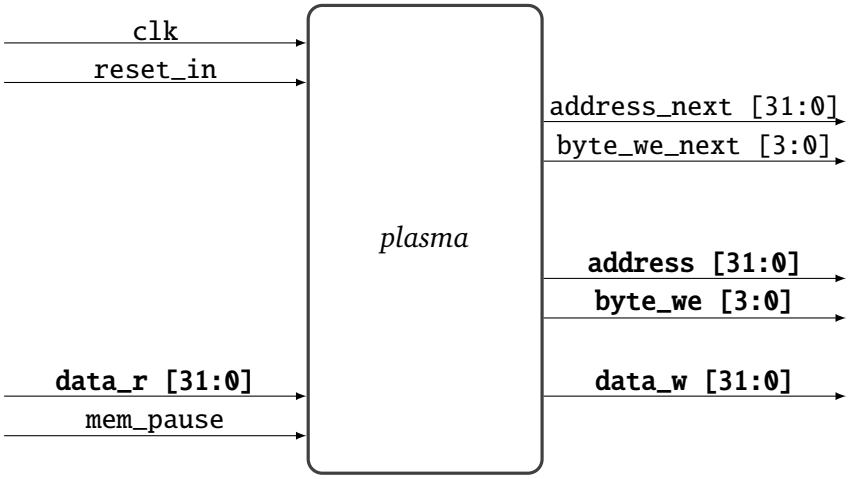


Abbildung 3.1: Externe Schnittstelle des *plamsa*-Kerns [Rhoads, 2013]

Neben dem Takt- und Reset-Eingang sind die Signale in zwei Gruppen gegliedert. Die fettmarkierten Signale stellen ein theoretisches Minimum für eine Speicheranbindung dar. Diese Signale sind notwendig, damit auch ein ideales Speichermodell angesprochen werden kann. Die anderen Signale sind für den Pufferspeicher und den dahinter folgenden DRAM vorgesehen. Die `_next`-Signale leiten die Informationen über die im nächsten Zugriff folgenden Werte der `address` und `byte_we` Leitungen, um den Pufferspeicher entsprechend vorzuladen. Das Signal `mem_pause` wird aktiviert, wenn der Pufferspeicher einen Fehlzugriff aufweist und höherliegende Ebenen anfragen muss. Diese drei Signale konnten bei einem idealen Speicher ignoriert werden.

Der *plasma* besitzt nur eine Adressleitung (`address`), die sowohl Instruktions- als auch Datenspeicher adressiert. Die gilt auch für den Dateneingang (`data_r`), der Instruktionen sowie Datenwerte liefert. Damit arbeitet dieser Kern mit einem reinen Speichermodell von [Newmann, 1945]. Der Datenausgang `data_w` wird, in Zusammenarbeit mit einem Maskensignal `byte_we`, für Schreiboperationen verwendet. Die Maskierung ist notwendig, da die Datenbreite des *plasma*-Kerns 32 Bit beträgt, wohingegen der Speicher byteweise angesprochen werden kann. Der MIPS I Befehlssatz sieht drei grundlegende Speicherzugriffsbefehle vor, die jeweils für das Schreiben und für vorzeichenbasiertes sowie vorzeichenloses Lesen gelten:

Tabelle 3.1: Speicherzugriffsbefehle des MIPS I Befehlssatzes und die dazugehörigen `byte_we` Werte

Datentransfer	Schreiben	Lesen	byte_we
kein Zugriff	-	-	0000
8 Bit	SB: *Store Byte*	LB: *Load Byte*	0001
		LBU: *Load Byte Unsigned*	0001
16 Bit	SH: *Store Half*	LH: *Load Half*	0011
		LHU: *Load Half Unsigned*	0011
	SWL: *Store Word Left*	LWL: *Load Word Left*	1111
	SWR: *Store Word Right*	LWR: *Load Word Right*	0000
32 Bit	SW: *Store Word*	LW: *Load Word*	1111

Es können also sowohl ganze Wörter mit 32 Bits als auch Halbwörter mit 16 Bits und einzelne Bytes mit 8 Bits übertragen werden. Bei einer vorzeichenbehafteten Variante werden die Teilwörter zwar auf die gesamte Breite von 32 Bits erweitert, der Speicherzugriff ist in diesen Fällen jedoch auf die jeweilige Breite begrenzt. Bei einer Reihe von speziellen Befehlen (mit *Left* und *Right*) wird berücksichtigt, dass die Adresse nicht zwangsläufig ein Vielfaches von 32 sein muss, sodass die Daten versetzt (*unaligned*) im Speicher abgelegt werden. Die Implementierung dieser Zugriffe und insbesondere die Nutzung durch den Compiler brachten allerdings einige Probleme mit sich. Beim *plasma*-Kern wurde aus patentrechtlichen Gründen auf eine Umsetzung verzichtet, sodass die *Left*-Befehle mit normalen Wortzugriffen gleichgestellt wurden und die *Right*-Befehle ignoriert wurden. Im Abschnitt 4.5 wird die Portierung der Software beschrieben und auf die Nutzung dieser Befehle eingegangen.

Interne Architektur

In der internen Architektur des *plasma* lassen sich klassische Bestandteile eines Prozessors erkennen, wie Abbildung 3.2 veranschaulicht.

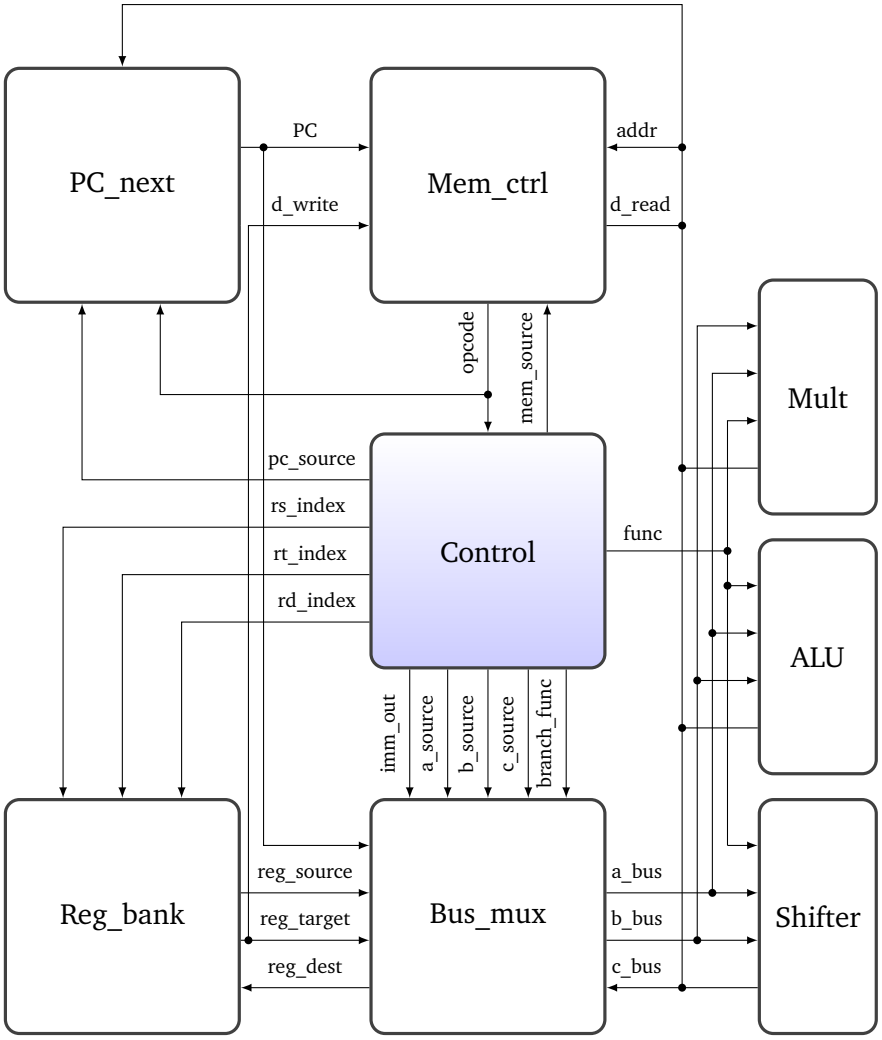


Abbildung 3.2: Interner Aufbau des *plasma*-Kerns [Rhoads, 2013]

Das Kontrolllogikmodul in der Mitte des Schaltbildes ist eng mit dem Datenpfad verknüpft. Der Datenpfad verfügt neben der Registerbank und Ausführungseinheiten über separate Module für Instruktionszeiger, Bus- und Speicheransteuerung. Die wichtigsten Daten und Steuersignale laufen im Bussteuerungsmodul zusammen und werden dort entsprechend weitergeleitet. Der *plasma* kann für genau 2 oder

mehr als 2 Pipelinestufen konfiguriert werden. Im zweiten Fall wird ein zusätzliches Modul für eine Reihe von Steuersignalen zwischengeschaltet, sodass diese entsprechend verzögert werden können.

Die Registerbank beinhaltet 32 frei ansteuerbare Register, wobei das Register `R0` nach Spezifikation einen konstanten Wert 0 liefert. Die Implementierung sieht mehrere Realisierungsmöglichkeiten vor, insbesondere Abbildungen auf mögliche FPGA-Bausteine. Außerdem verwaltet dieses Modul Teile der Unterbrechungslogik. Der Instruktionszeiger wird in einem separaten Modul verwaltet, das abhängig von einem Steuersignal aktuelle und künftige Adressen bereitstellt. Die aktuelle Adresse wird im Speicherkontrollmodul ausgewertet, wo auch alle externen Signale geschaltet werden. Der *plasma* verfügt über drei Operationseinheiten, ein separates Schiebemodul, einen Multiplikator und eine arithmetisch-logische Einheit. Die Modellierung der Funktionen erfolgt in nahezu allen Fällen auf Bitebene, insbesondere Additionen und Multiplikationen. Eine Multiplikation oder Division dauert mindestens 32 Takte, wobei das Ergebnis Bit für Bit errechnet wird. Währenddessen wird der Prozessor weitgehend angehalten. Es erfolgen keine Speicherzugriffe.

Insgesamt lässt sich feststellen, dass die Implementierung zwar modular erfolgt, die Funktionalität der Kontrolllogik aber in mehreren Fällen auf die Datenpfadmodule verteilt ist. Dadurch entstehen Rückkopplungen, die auf den ersten Blick nicht sichtbar sind. Dies erschwert wiederum Einarbeitungen eventueller Änderungen. Zudem ist die Modellierungsweise nicht einheitlich. Manche Module werden nahezu vollständig innerhalb eines VHDL-Prozesses implementiert. Alle Zwischenergebnisse werden in Variablen abgelegt. So ist es zwar möglich, die sequenzielle Ausführung nachzuverfolgen, die Werte der Variablen können aber nicht im Signalfenster des Simulators dargestellt werden. Außerdem wird bei der Modellierung nicht zwischen sequentieller und kombinatorischer Logik unterschieden, sodass während der Simulation ein VHDL-Prozess sowohl auf eine Änderung des Takts als auch auf eine Änderung der Eingangssignale der Logik reagiert. Trotz der Schwächen in der Modellierung implementiert der *plasma* den MIPS I-Befehlessatz nahezu vollständig und ermöglicht so die Ausführung entsprechend compilierter Applikationen.

Einsatz in der Simulation

Der unveränderte *plasma* wurde für erste Experimente ausschließlich innerhalb einer Simulation verwendet. Dabei wurden keine Implementierungsfehler festgestellt. Es hat sich allerdings gezeigt, dass eine Ausführung von Applikationen, die nur kleine Eingangsdateien (8 mal 8 Bildpunkte kleine Bilder) zu verarbeiten hatten, relativ viel realer Simulationszeit (einige Minuten) beanspruchte. Die Bearbeitung von Referenzbildern (512 mal 512 Bildpunkte) dauerte selbst mit einem idealen Speicher über zwei Wochen. Eine Ursachenanalyse brachte folgende Ergebnisse:

- Die Simulationssoftware konnte nur auf einem Prozessorkern des Arbeitsservers ausgeführt werden [Mentor Graphics Corporation, 2013]. Eine Parallelisierung der Kernfunktionen eines Simulators würde neben einer möglicher Datenabhängigkeit keine Reproduzierbarkeit mehr zulassen. Für die reale Simulationszeit kam es also ausschließlich auf die Taktfrequenz eines Kerns an.
- Die Modellierungsweise des *plasma*-Kerns ließ den Simulator viele VHDL-Prozesse auswerten, ohne dass es für große Teile der darin implementierten Logik relevant war.
- Eine Implementierung der Additions-, Schiebe- und Multiplikationsoperationen auf der Bitebene erforderte vom Simulator eine entsprechend längere Bearbeitungszeit.

Während der Portierung mussten einige Softwarefehler behoben werden. Der Umstand, dass selbst kleine Testbilder minutenlang simuliert werden mussten, erschwerte die Arbeit spürbar. Außerdem erschien die fehlende Aufteilung in Instruktion- und Datenadresse wenig repräsentativ für moderne Systeme. Daher wurde beschlossen, einige Grundkonzepte aus der Implementierung der Module des *plasmas* beizubehalten, die grundlegende Architektur aber tiefgreifend zu verändern.

3.1.2 Veränderter *plasma*

Abbildung 3.3 zeigt die veränderte Schnittstelle des CPU-Kerns.

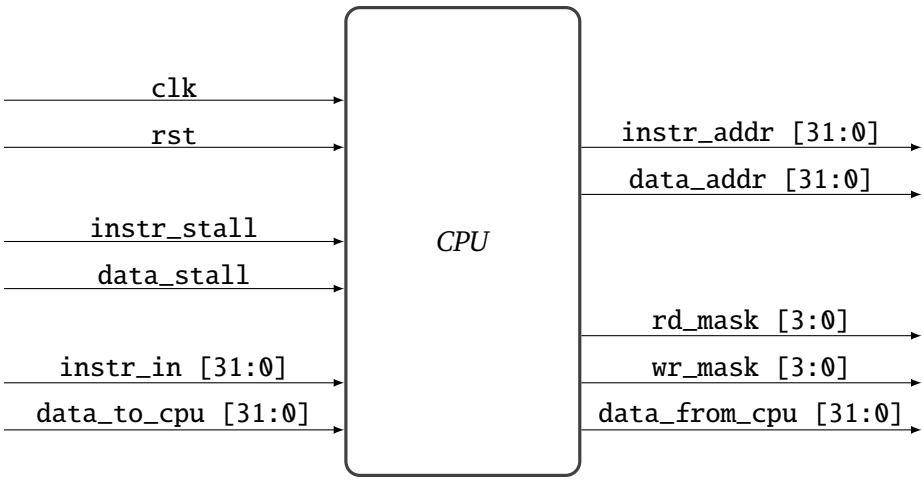


Abbildung 3.3: Externe Schnittstelle des veränderten CPU-Kerns

Die wesentliche Veränderung betrifft die Aufteilung in Instruktions- und Datenspeicher, die entsprechende Adressleitungen (`instr_addr` und `data_addr`), Datenleitungen (`instr_in`, `data_to_cpu`, `data_from_cpu`) und Fehlzugriffsleitungen (`instr_stall`, `data_stall`) erforderten. Auf eine Bereitstellung der Daten im Voraus mittels `_next`-Signale wurde verzichtet. Die Funktion des Signals `byte_we` wurde auf zwei Signale aufgeteilt, wobei so ein Lese- und ein Schreibzugriff so separat angesteuert werden konnten. Diese Veränderung verhinderte eine permanente Auswertung des `data_addr`-Signals durch den Pufferspeicher. Auch ein Lesezugriff musste entsprechend eingeleitet werden, um unnötige Aktualisierungsvorgänge in der Datenkomponente des Pufferspeichers zu vermeiden.

Interne Architektur

Der Datenpfad weist große Unterschiede zu *plasma* auf. Eine vereinfachte Darstellung mit allen Untermodulen und wichtigsten Datenleitung ist in Abbildung 3.4 dargestellt. Die Signale der externen Schnittstelle sind rot markiert oder hinterlegt. Einige Konzepte, insbesondere solche, die sich auf einzelne Komponenten beziehen, wurden vom *plasma* übernommen. Die meisten Leitungen der externen Schnittstelle werden weiterhin durch ein Speicherverwaltungsmodul `MEM_CTRL` verwaltet. Die jeweiligen Adressen werden hingegen direkt im Datenpfad generiert. Die Instruktionsadresse wird aus dem entsprechenden Register heraus geleitet und die Ausgabe der Funktionseinheiten kann als Datenadresse interpretiert werden. An dieser Stelle ist ersichtlich, dass eine permanente Auswertung der Werte dieser Leitung als eine Adresse zu ständigen Datentransfers im Pufferspeicher führen würde.

Im Unterschied zu *plasma* verfügt der veränderte CPU-Kern über eine feste Anzahl an Pipelinestufen. Außerdem wurde zu den drei Funktionseinheiten eine separate Vergleichseinheit hinzugefügt, die im Falle eines entsprechenden Befehls die Auswertung vornimmt. Der Einsatz der Pipeline macht eine zusätzliche Logik für Konfliktlösungen notwendig. Ein Konflikterkennungsmodul wurde eingefügt. Die Kontrolllogik kann auf verschiedene Arten von Konflikten durch gezielte Sperrung der Pipelineregister reagieren. Zudem ist es möglich, das Instruktionszeigerregister mit Nullen zu füllen. An dieser Stelle befindet sich ein Übergang von Verzögerungsursachen, die eindeutig auf den Speicher zurückzuführen sind, hin zum Prozessor. Die Ursachen für Unterbrechung des Pipelineflusses kann verschiedene Ursachen haben und ist damit nicht eindeutig zuordenbar:

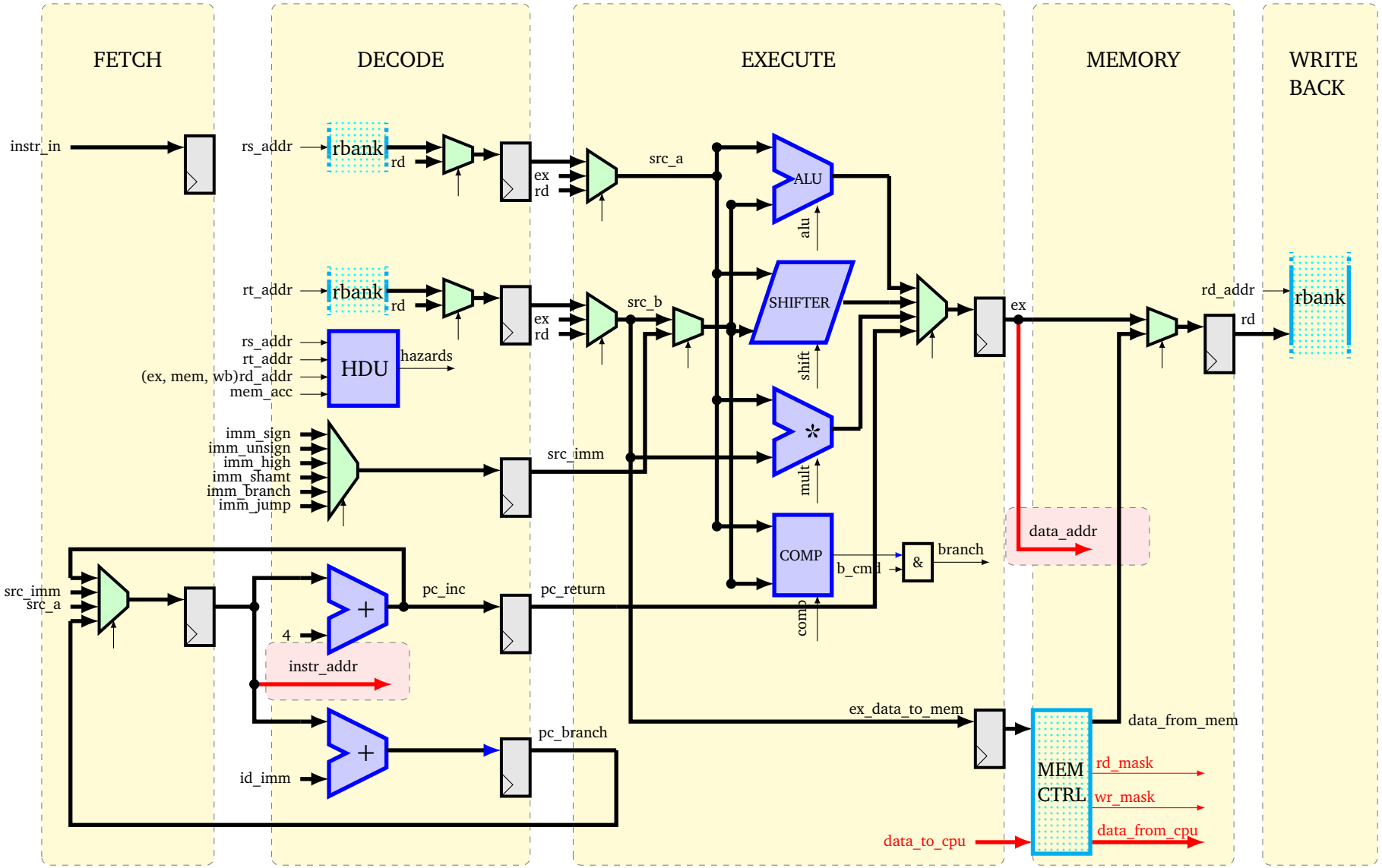


Abbildung 3.4: Datenpfad des veränderten CPU-Kerns

1. Fehlzugriff beim Instruktions- oder Datenspeicher (eines der Signale `instr_stall` oder `data_stall` ist gesetzt) führen zu einem kompletten Stopp der Pipeline.

2. Datenkonflikte: (1) Daten aus dem Speicher, die erst in der Speicherzugriffsstufe zur Verfügung stehen, werden bereits in der Decodierungsstufe benötigt. (2) Das Sprungziel wird aus einem Registerwert, der noch berechnet wird, entnommen. Dieser Konflikt führt zum Stopp der ersten Pipelinestufen, wobei im zweiten Fall eine „`no operation`“ an die Ausführungsphase weitergereicht wird.

3. Wenn das Ergebnis der Multiplikation gebraucht wird, ohne dass die Operation abgeschlossen ist, führt das zu einem kompletten Halt der Pipeline.

Alle diese Situationen können auch kombiniert auftreten, wobei die zwei zuletzt genannten durch eine geschickte Anordnung der Maschinenbefehle durch den Compiler in bestimmten Fällen vermieden werden können. Wenn man das gesamte System betrachtet, wird an diesem Punkt deutlich, dass die Laufzeit auch mit einem idealen Speicher noch weiter verbessert werden kann. Alle Verzögerungen, die ein reales Speichersystem verursacht, können durch das erste Szenario beschrieben werden. Es kommt letzten Endes darauf an, welchen Anteil diese Verzögerungen an der gesamten Ausführungszeit haben.

Die einzelnen Komponenten der veränderten CPU weisen in den meisten Fällen die Implementierungsansätze von *plasma* auf. Es wurde allerdings bei der Modellierung darauf geachtet, dass die kombinatorische Logik, soweit es möglich war, außerhalb eines VHDL-Prozesses zu implementieren und sie durchgehend von sequentiellen Anteilen zu trennen. Die wichtigsten Veränderungen betrafen die Ausführungseinheiten. Insbesondere die arithmetisch-logische Einheit und der Multiplikator bekamen zwei Implementierungsvarianten (zwei *architectures*): Eine wurde in erster Linie für Simulationen konzipiert, während die andere für eine Synthese auf einem FPGA abzielte. Insbesondere Additions- und Multiplikationsroutinen wurden in der Simulationsversion durch Funktionsaufrufe externer Bibliotheken umgesetzt.

Über die Auswirkungen der einzelnen Veränderungen für sich kann nur spekuliert werden, da die interne Verarbeitung des Simulators nicht einsehbar ist. Im Endergebnis konnten die reale Simulationszeit desselben Maschinencode sowie Eingangsdaten von 2 min 33 sec auf 48 sec reduziert werden. Für große Eingangsdaten bedeutete dies eine Reduktion von mehreren Wochen auf ein paar Tage. Diese im Vergleich zum *plasma* beschleunigte Simulationsausführung eröffnete auch Möglichkeiten, umfangreichere Applikationen zu testen. Bei einem Videokomprimierungsalgorithmus stellte sich allerdings heraus, dass während der Ausführungszeit nahezu ausschließlich Fließkommaoperationen durchgeführt wurden. Die Softwareemulation dieser Operationen verlangsamte allerdings die Ausführung. Ein Hardwaremodul für diese Operationen (eine FPU) sollte für weitere Beschleunigung sorgen.

3.1.3 FPU

Bereits in der ersten Version der MIPS-Spezifikation wurden Fließkommazahlenbefehle definiert [MIPS Technologies Inc., 2013]. Während der Übersetzung durch einen Compiler wird je nach Konfiguration entschieden, ob diese Befehle auch verwendet werden. Wenn keine Hardwareeinheit vorhanden ist, werden die Fließkommaoperationen in Softwareroutinen umgewandelt. Anderenfalls müssen diese Befehle entsprechend an die FPU weitergeleitet werden. Deshalb betraf die Integration einer FPU in erster Linie den Decoder. Im Datenpfad mussten nur zusätzliche Daten- und Kontrollverbindungen umgesetzt werden. Abbildung 3.5 zeigt den Datenpfad des Moduls, die Verbindungsleitungen zum CPU-Datenpfad sind rot hinterlegt.

Das Modul verfügt über eine arithmetisch-logische Einheit, die die eigentlichen Fließkommaoperationen ausführt, einen Vergleicher, der ebenfalls mit Fließkommazahlen umgehen muss, und eine separate Registerbank. Der Befehlssatz unterstützt sowohl einfache als auch doppelte Genauigkeit. Fließkommazahlen mit doppelter Genauigkeit werden mithilfe von zwei Registerzellen je 32 Bit repräsentiert. Die

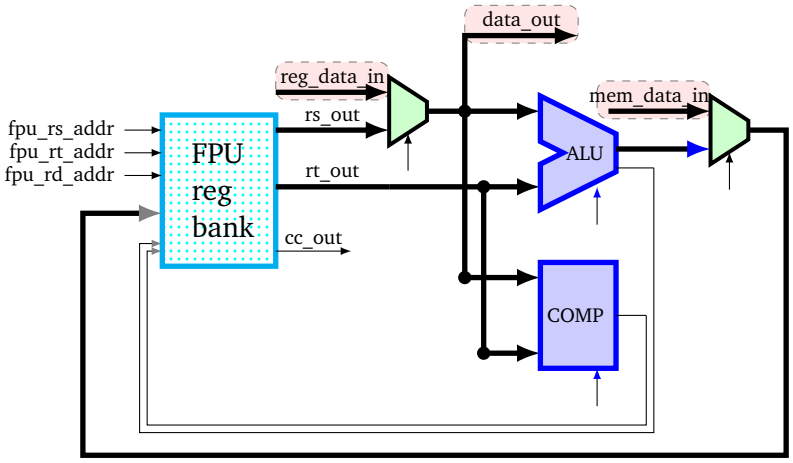


Abbildung 3.5: Datenpfad der FPU [Reuter, 2015]

Registerbank verfügt über 32 solcher frei adressierbaren Plätze. Beim Datenaustausch zu und von der FPU übernimmt der Compiler die Adressanpassung. Es werden in der Regel zwei Registeradressen für eine Zahl reserviert. Insbesondere bei Speicherzugriffen, die direkt zu der FPU führen, ist die Platzierung der beiden Teile der Zahl entscheidend.

Die eigentliche Ausführung der Fließkommaoperationen wird in der `ALU` durchgeführt. Des Weiteren stellt diese Komponente alle Konvertierungsfunktionen bereit, um Daten zwischen der FPU- und der CPU-Registerbank sowie zwischen Genauigkeitsstufen austauschen zu können. Diese Funktionen wurden über zwei Ansätze realisiert:

- Eine VHDL-Fließkommabibliothek stellt Funktionen zur Verfügung, um die entsprechenden Operationen durchführen zu können.

- Der Simulator verfügt über eine externe Schnittstelle, über die eine C-Bibliothek mit entsprechenden Funktionen angebunden werden kann. Die Ausfürhung erfolgt dann direkt auf der Hardware des Servers.

Der Einsatz der FPU war nur für Simulationen konzipiert, da die reale Ausführungszeit von softwareemulierten Fließkommaoperationen auf einem FPGA nur einige Sekunden in Anspruch nahm. Die reale Simulationszeit war dagegen von entscheidender Bedeutung. Auf den Einsatz der FPU wird im Abschnitt 4.5.3 bei Portierung einer Testapplikation näher eingegangen.

Eine vorläufige Version des Prozessorkerns mit vier Pipelinestufen und der integrierten FPU wurde als ein quelloffenes Projekt in [Schoenberger und Reuter, 2015] veröffentlicht.

3.2 Mehrkernsystem

Seit den ersten integrierten Schaltungen war eine Größe maßgebend für die Beschreibung des Systems: die Taktfrequenz. Insbesondere für Prozessoren stellte die Zunahme der Taktfrequenz einen der wichtigsten treibenden Kräfte für Innovationen dar. Anfang der 2000-er Jahre stellte sich allerdings heraus, dass der mit der Taktfrequenz steigende Energieverbrauch und damit die Wärmeentwicklung einen Wert erreicht hatten, der nicht mehr mit vertretbarem Aufwand zu regulieren war. Dies betraf allerdings nicht die Integrationsdichte. Daher löste ein Wachstum der Anzahl von Prozessorkernen die Taktfrequenzzunahme ab. Ein Mehrkernsystem ist seitdem die Standardausführung für Desktop- und Serversysteme sowie etwas später für mobile und Multimediageräte.

3.2.1 Topologie

Der Umstieg von einem Einkernsystem zu einem Mehrkernsystem ist im Bereich der Hardware im Vergleich zur Software weniger problematisch. Die Herausforderungen beim Parallelisieren von Applikationen werden in Abschnitt 4.4 erläutert, beim Verbund von mehreren Prozessorkernen gilt es in erster Linie zu entscheiden, wie die Kerne miteinander kommunizieren sollen. Eine Möglichkeit besteht darin, die Kerne an einen gemeinsamen Speicher anzubinden (*shared memory*) und die Kommunikation so über diesen Speicher zu ermöglichen.

Eine weitere Möglichkeit besteht darin, die Kerne über spezielle Vermittlungsknoten (und im Weiteren *router*) miteinander zu verbinden. Diese Konstellation hat den Vorteil, dass die Nachrichtenübertragung über das Routernetzwerk verläuft, sodass unterschiedliche Komponenten wie Digitale Signalprozessoren, GPUs und weitere Module in einem Netzwerk miteinander kommunizieren können. In diesem Fall spricht man von einem NoC (***N**etwork-**o**n-**C**hip*). Auch der Speicher kann eine solche Komponente sein (*distributed memory*). In diesem Fall ist es nur erfoderlich, dass der Speicher eine Schnittstelle zum Router implementiert. Allerdings ist davon auszugehen, dass dieser Router relativ großen Netzwerkverkehr verursachen wird.

Die Thematik der Mehrkernprozessoren und NoCs ist sehr umfangreich und Gegenstand aktueller Forschungsaktivitäten, sodass hier nur ein kleiner Ausschnitt erläutert werden kann. In dieser Arbeit geht es um Speicher. Für eine Analyse der Speichernutzung auf einem Mehrkernsystem dient ein Mehrkernsystem in erster Linie als Testumgebung. Eine Variation der unterschiedlichen Mehrkernsystemkonfigurationen könnte Teil einer weiterführenden Arbeit werden. Als erster Ansatzpunkt wurde ein Mehrkernsystem (Abbildung 3.6) implementiert, das einfach modellierbar ist und viele Möglichkeiten bietet, die Speicherschnittstelle beobachten zu können.

Jeder Kern ist über einen Abzweiger an den gemeinsam genutzten Speicher angebunden. Der Abzweiger wertet die Datenadresse aus. Falls diese Adresse einem der vordefinierten Werte gleicht, leitet er die Datensignale an den jeweiligen Router weiter. Die Instruktionsadresse und die Instruktionsdaten werden direkt mit dem jeweiligen Kern verbunden. Auch sonst verfügt der Speicher über so viele Schnittstellen, wie es Kerne im Netzwerk gibt. Die Router sind in einem Kreis miteinander verbunden und verfügen über Identifikationsdaten.

Die Kerne können in dieser Struktur auf zwei Weisen miteinander kommunizieren: über den gemeinsam genutzten Speicher und über das Routernetzwerk. Große Datenmengen müssen somit nicht über das Netzwerk verschickt werden. Die spezielle Kennung für jeden Kern macht es möglich, ihn für spezielle Operationen zu reservieren. Die Nachrichtenübertragung im Routernetzwerk braucht dabei weniger Zeit im Vergleich zu einer Übertragung im Speicher, da die Daten bei einem Schreibzugriff erst die verschiedenen Ebenen des Speichersystems passieren müssen. Außerdem müssen keine Datenabhängigkeiten bei einer Kommunikation über das Routernetzwerk aufgelöst werden. Eine solche Netzwerkstruktur kann allerdings nur für eine kleine Anzahl an Prozessorkernen realisiert werden. Die Untersuchungen für Mehrkernsysteme bieten erste Ansatzpunkte für eine solche Art von Lösungen.

3.2.2 Router

Die wichtigste Komponente eines Netzwerks ist der Verteilungsknoten, der Router. Alle Router des Mehrkernnetzwerks, das bei dieser Arbeit verwendet wurde, sind wie in Abbildung 3.7 aufgebaut.

Der Router verfügt über drei Register. Nur zwei davon können direkt vom Prozessorkern angesprochen werden. Zusammen mit den Daten wird auch die Zieladresse über das Netzwerk übertragen. Der Router vergleicht diese mit seiner eigenen. Wenn die Adressen übereinstimmen, werden die Daten in das entsprechende Register geleitet. Anderenfalls werden sie an den nächsten Knotenpunkt weitergegeben. Über die Kontroll- und Statusregister kann der Prozessorkern darüber informiert werden, dass Daten empfangen wurden oder der Inhalt des Datenregisters an die angegebene Adresse verschickt werden

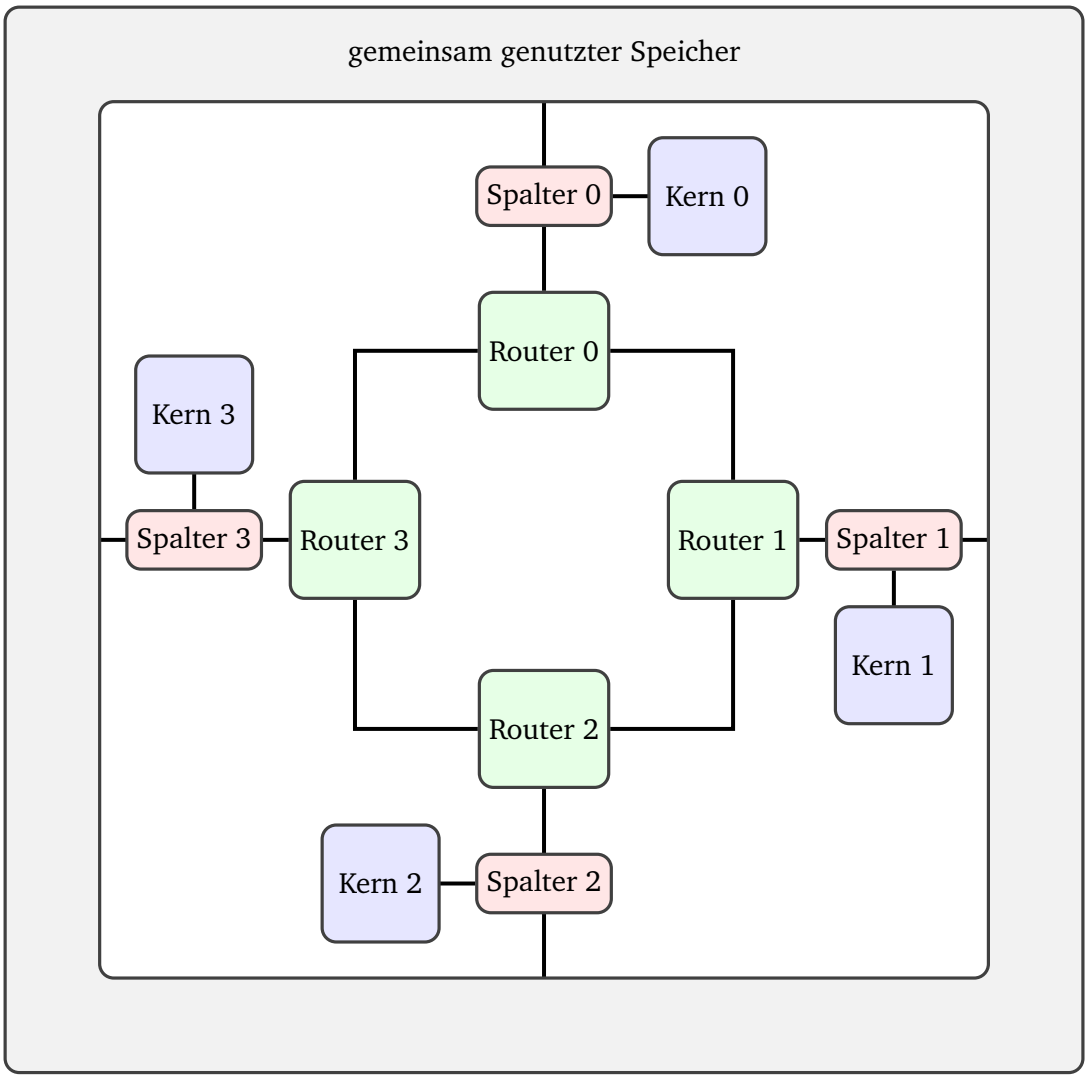


Abbildung 3.6: Struktur des Mehrkernnetzwerks, ein Beispiel mit 4 Kernen

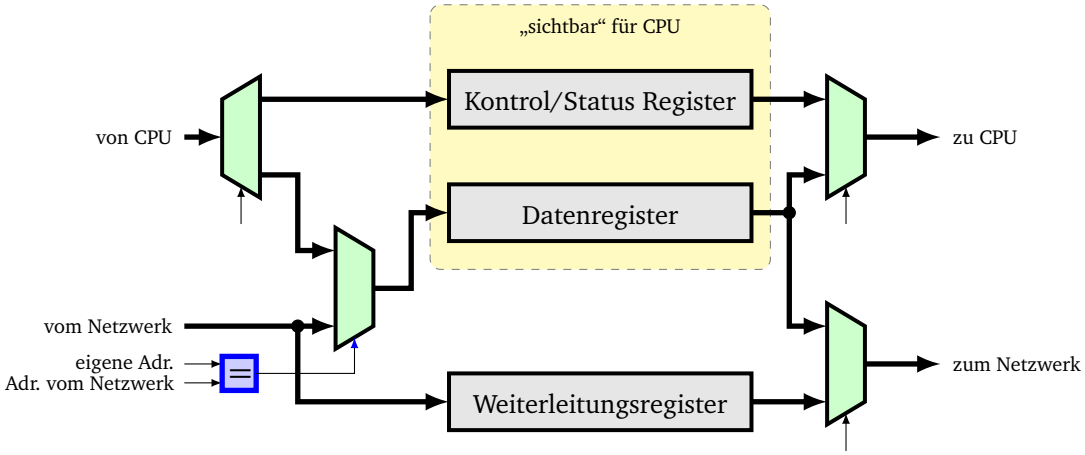


Abbildung 3.7: Struktur des Routers

soll. Außerdem enthält das Statusregister eine einmalige Kennung für jeden Prozessorkern, sodass er sich über dieses Bitfeld identifizieren kann.

Diese Implementierung in Kombination mit der im vorigen Abschnitt präsentierten Topologie benötigt kein Routing. Da es nur ein Datenregister gibt und da keinerlei Maßnahmen für Pufferung zur Verfügung stehen, kann diese Lösung nur für Anwendungen zum Einsatz kommen, die nur sehr wenige Netzwerkübertragungen benötigen. Die Routermodellierung ist synthetisierbar und kann in einem FPGA eingesetzt werden.

3.3 Speichermodell

Die wichtigste Komponente für diese Arbeit war das Speichermodell. Um einen Effekt durch die Stapelung von DRAM-Schichten feststellen zu können, ist es allerdings zunächst erforderlich, eine Vergleichsbasis zu bilden. Insbesondere beim Vergleich von Laufzeiten ist die Bezugsgröße entscheidend. Da die Untersuchungen ausschließlich simulativ durchgeführt wurden, ist zudem ein möglichst realitätsnahes Modell von großer Bedeutung. In den folgenden Abschnitten werden zunächst die Modelle für einen idealen und einen DRAM-Speicher vorgestellt. Der Pufferspeicher ist derzeit eine unverzichtbare Komponente in modernen Systemen. Daher wird auch seine Modellierung im darauf folgenden Abschnitt erläutert. Abschließend wird die Modellierung des 3D-DRAM sowie die untersuchte Konfiguration beschrieben.

3.3.1 Referenzmodell

Um eine Vergleichsbasis bilden zu können, muss zunächst der Einfluss des Speichers auf die Laufzeit festgestellt werden. Denn selbst mit einem idealen Speicher benötigt eine Applikation eine bestimmte Anzahl an Ausführungszyklen. Diese Anzahl lässt sich nur extrahieren, wenn der Speicher keine Verzögerungen verursacht, d. h. wenn der Speicher ideal ist.

Idealer Speicher

Ein idealer Speicher kann wie folgt definiert werden:

> *Ein idealer Speicher besitzt eine unbegrenzte Kapazität und kann die Daten ohne jegliche Verzögerung speichern und lesen.*

Die erste Anforderung, die ein idealer Speicher aufstellt, kann auf einem realen System, einem Server, auf dem die Simulation ausgeführt wird, nicht erfüllt werden. Wenn ein Modell allerdings alle genutzten Daten umfassen kann, besteht für die jeweilige Applikation faktisch kein Unterschied zu einer unbegrenzten Kapazität. Die erste Anforderung kann also auf eine applikationsspezifische Größe reduziert werden. Eine fehlende Verzögerung lässt sich ebenfalls in einem Simulationsmodell ohne Weiteres realisieren.

Der ideale Speicher sollte in erster Linie dazu genutzt werden, nur die vom DRAM-Speicher verursachte Zeit aus der Gesamtlaufzeit zu extrahieren. Dazu musste die Speicherverzögerung um den Pufferspeicheranteil bereinigt werden. Abbildung 3.8 zeigt die dazu nötigen Messkonfigurationen.

Bei der ersten Messung kann die Anzahl der Zyklen erfasst werden, die eine Applikation für die Ausführung braucht. Diese Anzahl kann für eine beliebige Taktfrequenz verwendet werden. Bei der zweiten Messung wird die tatsächliche Ausführungszeit gemessen, wobei der Pufferspeicheranteil hier mit Kenntnis der idealen Laufzeit berechnet werden kann.

Für diese beiden Messungen mussten ideale Speicher modelliert werden, bei denen die Kernimplementierungsansätze gleich waren. Die Speicherfunktion wurde durch ein Datenfeld (*array*) realisiert. Die konkrete Implementierung erfolgte in der Hardwarebeschreibungssprache VHDL. Da die Simulationen mithilfe des Simulators von Mentor Graphics - ModelSIM - durchgeführt wurden, fanden spezielle

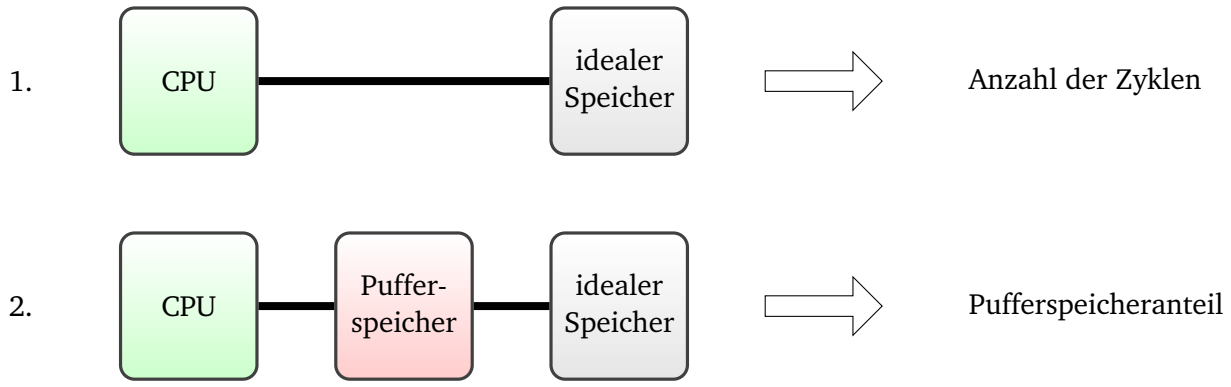


Abbildung 3.8: Messung der idealen Laufzeit und die Extraktion des Pufferspeicheranteils

Modellierungskonzepte Einzug in den Quellcode. Ein adressierbares Datenfeld kann in VHDL auf zwei Arten deklariert werden: als ein Signal oder eine Variable. Ein deklariertes Signal kann zum Signalfenster hinzugefügt werden, und dessen Wertänderung kann so nachverfolgt werden. Dieser Implementierungsansatz hatte insbesondere während der Portierung von Software einen großen Vorteil, da die Änderung der Speicherwerte direkt nachverfolgt werden konnten. Allerdings stellte sich schon bald heraus, dass dieser Ansatz nicht nur die reale Simulationszeit verlängerte, sondern darüber hinaus ausschließlich für kleine Datenfelder verwendet werden konnte. Jedes Bit dieses Datenfeldes konnte 9 Zustände annehmen (ausgehend vom Datentyp `std_logic`) und beanspruchte dementsprechend viel Speicherplatz auf dem Server, obwohl nur zwei Zustände ('0' und '1') relevant waren. Außerdem war der Kernel des Simulators nicht in der Lage, große Mengen solcher Datenfelder vom Betriebssystem des Servers reservieren zu lassen. Dies führte dazu, dass die Verarbeitung von nur kleinen Datenmengen durch eine Applikation simuliert werden konnte.

Eine andere Möglichkeit dafür, die Datenfelder in VHDL zu modellieren, boten Variablen. Genau für solche Zwecke, also insbesondere für die Modellierung großer Speicher, ist in ModelSIM das Konzept der `shared variable` vorgesehen. Der übliche Einsatz von Variablen in VHDL erlaubt es nur, sie innerhalb eines Prozesses zu deklarieren und zu benutzen. Sie können während der Simulation nicht im Signalfenster betrachtet werden und müssen jedes Mal, wenn der zugehörige Prozess ausgewertet wird, neu initialisiert werden. Das Konzept der `shared variable` und insbesondere die Interpretation durch ModelSIM erlaubt dagegen die Deklaration und Benutzung solcher Variablen in mehreren Prozessen. Sie behalten ihren Wert. Diese Werte können zwar nach wie vor nicht im Signalfenster angezeigt werden, die Größe der Datenfelder stellt jedoch keinen limitierenden Parameter mehr dar.

In dieser Arbeit wurden beide Implementierungsmöglichkeiten verwendet. Die erste Variante wurde in der Aufbauphase genutzt, während die Endergebnisse ausschließlich durch den Einsatz der `shared variable` generiert werden konnten. Die Datenfelder wurden also in beiden Komponenten der idealen Speicher auf die gleiche Art und Weise realisiert. Die Schnittstellen sowie die Zellengröße markieren Unterschiede, wie in Tabelle 3.3.1 aufgelistet wird.

Tabelle 3.3.1: Konfiguration der idealen Speicherkomponenten

Einsatz	Daten	Anzahl der Leseschnitt-stellen	Anzahl der Schreib-schnittstel-len	Adressraum	Zellengröße [Bit]
ideale Laufzeit	`shared variable`	2	1	`0 - 0x0200 0014`	8
ideale Laufzeit mit Pufferspeicher	`shared variable`	1	1	`0 - 0x0020 0000`	1024

Der wesentliche Unterschied bestand also in der Zellengröße, die bei der Komponente, die an den Pufferspeicher angeschlossen war, dem Datenpaket entsprach (*cache line*). Wegen der Trennung von Daten- und Instruktionszugriffen benötigte das entsprechende Speichermodell zwei Leseschnittstellen. Die Anordnung der Bytes innerhalb eines Wortes entsprach dem LE-Konzept (***L**ittle **E**ndian*), sodass die Daten bei einem Zugriff wie folgt angeordnet wurden:

3.3.1 Zuordnung der Bytes am Beispiel eines Lesezugriffs

```
1     case mask is
2       when "0001"   =>
3         data(31 downto 24)  <= memory(i_addr  );
4         data(23 downto 16)  <= memory(i_addr  );
5         data(15 downto  8)  <= memory(i_addr  );
6         data( 7 downto  0)  <= memory(i_addr  );
7
8       when "0011"   =>
9         data(31 downto 24)  <= memory(i_addr  );
10        data(23 downto 16)  <= memory(i_addr + 1);
11        data(15 downto  8)  <= memory(i_addr  );
12        data( 7 downto  0)  <= memory(i_addr + 1);
13
14      when "1111"   =>
15        data(31 downto 24)  <= memory(i_addr  );
16        data(23 downto 16)  <= memory(i_addr + 1);
17        data(15 downto  8)  <= memory(i_addr + 2);
18        data( 7 downto  0)  <= memory(i_addr + 3);
```

Bei Teilwortanfragen wurden dennoch ganze Wörter zurückgegeben, um es der Prozessorlogik zu überlassen, welche Teile übernommen werden sollen. Insbesondere im Hinblick auf den Pufferspeicher war eine feste Zuordnung der Bits 31 bis 24 auf die Zugriffsadresse (Zeilen 3, 9 und 15) vorteilhaft.

Die Implementierung des idealen Speichers erforderte einen vergleichsweise geringen Aufwand. Außerdem wurde seine Einbindung durch weitere Scripte (siehe 4.3.3) unterstützt. Eine völlig andere Stufe der Komplexität stellt allerdings das Referenzmodell des DRAM-Speichers dar.

DDR2-DRAM

Um einen Vergleich zwischen zwei Systemen durchführen zu können, ist es zunächst erforderlich, die Vergleichsgröße festzulegen. Im Idealfall sind die Vergleichsbedingungen identisch und nur diese Größe wird variiert. Bei einer solchen Konstellation lässt sich direkt zwischen zwei Alternativen entscheiden. Bei komplexen, nicht linearen Systemen liegt die Herausforderung darin, diese Vergleichsgröße einzugrenzen. Bei einem gestapelten DRAM stellt sich die Frage, zu welcher Komponente des Speichersystems

dieser DRAM eine Alternative darstellen soll und ob es überhaupt alternativ eingesetzt werden muss. So präsentiert [Dong et al., 2010] beispielsweise ein Konzept, bei dem sowohl ein gestapelter als auch ein konventioneller DRAM gemeinsam genutzt werden. Dei [Lee et al., 2015] agiert der gestapelte DRAM als preisgünstiger Pufferspeicher und tritt damit in Konkurrenz zur üblichen SRAM-Technik. Eine abschließende Antwort in Bezug auf den alternativen Einsatz kann somit nur gegeben werden, wenn die restlichen Anforderungen wie Kosten, Energieverbrauch, Stückzahl usw. bekannt sind.

Um einen Ausgangswert in Bezug auf die gegenwärtige Technik zu haben, wurde bei dieser Arbeit ein DDR2-DRAM-Modell [Micron Technology Inc., 2013] des Halbleiterherstellers Micron Technologie verwendet. An dieser Stelle stellt sich allerdings direkt die Frage, warum nicht die neuere DDR3-Technik gewählt wurde. Ein Modell des DDR3-DRAM wird von Micron ebenfalls zur Verfügung gestellt. Es stellt ein realitätsnahes Verhaltensmodell realer Bausteine dar. Der wesentliche Unterschied zwischen den beiden Modellen besteht im Modellierungsumfang, der Auswirkungen auf den DRAM-Controller und die Berechnung der Verzögerungen hat. Im vorigen Abschnitt wurde bereits gezeigt, dass genügend Speicherkapazität modelliert werden konnte. Der wesentliche **limitierender Faktor** war die **reale Simulationszeit**. Diese Zeit unterschied sich zwischen den DDR2- und DDR3-Modellen um eine ganze Größenordnung. Es kam noch erschwerend hinzu, dass der Simulator das Modell nicht intern optimieren konnte. Wenn diese Option ausgewählt wurde, verhielt sich das Modell fehlerhaft. Die ganze Testumgebung musste mit diesem Speichermodell ohne jegliche Optimierung ausgeführt werden. Eine Simulation der größten Eingangsdaten dauerte bereits mit einem DDR2-DRAM-Modell mindestens zwei Wochen. Ein DDR3-Modell hätte diese Zeitspanne vermutlich auf mehrere Monate ausgedehnt.

Ein Vergleich mit DDR3-DRAM hätte zwar einen direkten Bezug zu aktueller Technologie, doch aus datentechnischer Sicht bestand der Unterschied zwischen DDR2 und DDR3 in kürzerer Zugriffszeit und größerer Datenmenge pro Zugriff. Der Effekt einer größeren Bandbreite insbesondere durch 3D-DRAM wurde aber bereits bei Woo [Woo et al., 2010] untersucht (Näheres im Abschnitt 2.2.2). Daher blieb nur die kürzere Zugriffszeit als Variable. Da aber eine kürzere Zugriffszeit sehr gut durch ein 3D-DRAM-Modell modelliert werden konnte, müssten die Ergebnisse der Untersuchung den Einfluss dieser Variable zeigen. In Bezug auf DDR3-DRAM hätte das nur bedeutet, dass die entsprechende Beschleunigung durch 3D-DRAM geringer ausfallen müsste.

Das Modell abstrahiert ein reales Bauteil. Das spiegelt sich auch in seiner Schnittstelle wieder, wie Abbildung 3.9 darstellt.

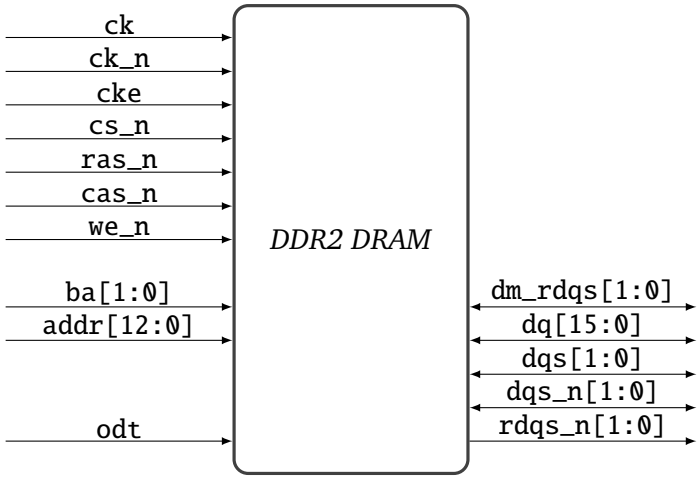


Abbildung 3.9: Schnittstelle des DDR2 DRAM Models

Die Schnittstelle entspricht dem Standard von JEDEC und muss auch entsprechend angesteuert werden. Das Modell ist in vielen Bereichen parametrisiert, sodass ein Verhalten von verschiedenen realen Lösungen von Micron modelliert werden kann. Die Implementierung ist im Wesentlichen aus einer Reihe von Verilog-Funktionen aufgebaut, die zwar keine Struktur abbilden, aber allein schon durch die Na-

mensgebung auf die einzelnen DRAM-Komponenten sowie auf Zugriffsprozesse schließen lassen. Die Speicherwerte werden in einem mehrdimensionalen Verilog-Datenfeld verwaltet. Im Gegensatz zum VHDL lassen sich diese Daten im Signalfenster anzeigen, ohne dass es Probleme damit gibt, den nötigen Speicherplatz auf dem Server zu reservieren. Die ModelSIM-Datenstruktur dieser Variable ist allerdings so groß, dass im Betrieb keine Anzeige möglich war, sodass es in der praktischen Nutzung keinen Unterschied zur Variable des VHDL-Models des idealen Speichers gab.

Das Modell konnte zwar direkt in eine Verilog-Umgebung integriert werden, die Ansteuerung musste allerdings standardkonform erfolgen, sodass es eines DRAM-Controllers bedurfe. Diese Komponente übernimmt unter anderem die Initialisierungsphase nach dem Einschaltvorgang, nimmt Lese- und Schreibbefehle der Applikationslogik entgegen und leitet sie über die Standardschnittstelle des DRAMs weiter. Sie sorgt dafür, dass alle Zeiten des Zugriffsprozesses eingehalten werden und die Schnittstelle zum Design flexibel bleibt. Das DDR2-DRAM-Modell war ein Teil eines möglichst realitätsnahen Testumfeldes. Da die Untersuchungen dieser Arbeit ursprünglich auf einem FPGA ausgeführt werden sollten, konnte ein entsprechender IP-Kern des Herstellers Xilinx verwendet werden.

Der Controller kann den DRAM mit bis zu 200 MHz ansprechen und vier oder acht Wörter lange Bursts übertragen. Außerdem verfügt er über Pufferkomponenten, um mehrere hintereinander folgende Zugriffe annehmen zu können [Xilinx, Inc., 2010]. Abbildung 3.10 zeigt die Schnittstelle des Controllers zum Applikationsdesign.

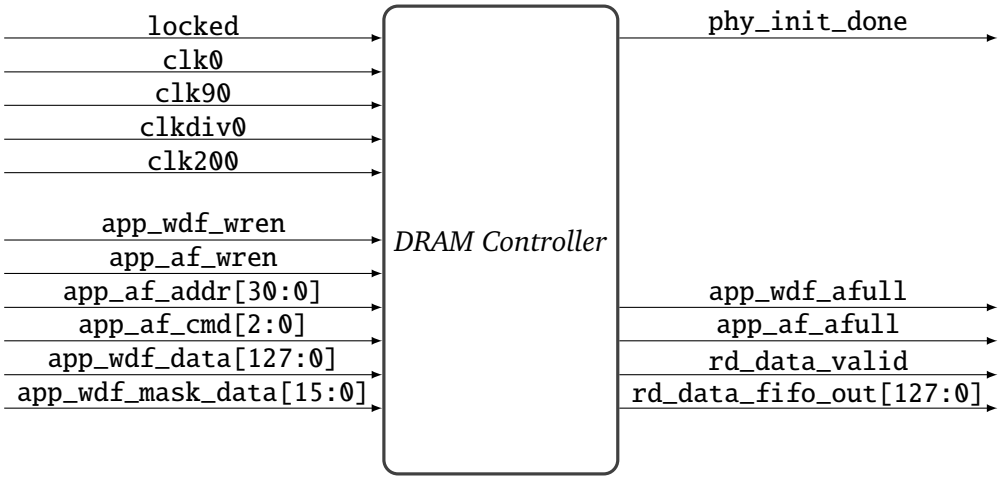


Abbildung 3.10: Schnittstelle des verwendeten DRAM-Controllers von Xilinx zum Applikationsdesign

Für die notwendigen Taktsignale musste ein zusätzliches PLL-Modul vorgeschaltet werden. Die Art der Ansteuerung konnte einem Referenzdesign entnommen werden, sodass die Applikationsschnittstelle an die Pufferspeicherschnittstelle angepasst werden konnte. Die übertrage Datenmenge pro Zugriff entsprach 1024 Bits.

3.3.2 Pufferspeicher

Die Modellierung des Pufferspeichers sollte wie bereits beim DRAM-Controller für einen künftigen FPGA-Einsatz geeignet sein. Zudem sollten sowohl ein System mit einem als auch mit mehreren Kernen getestet werden können. Der Pufferspeicher sollte also synthetisierbar und für Mehrkernsysteme erweiterbar sein. Es wurden zwei Ebenen des Pufferspeichers modelliert, wobei es auf der ersten Ebene eine Aufteilung in eine Instruktions- und eine Datenkomponente gab.

Eine der wesentlichen Unterschiede zwischen den beiden Komponenten des Pufferspeichers auf der ersten Ebenen war, dass der Instruktionsspeicher keine Schreibzugriffe vom Prozessor verarbeiten musste. Dies führte zu einer Vereinfachung der Kontrolllogik. Abbildung 3.11 zeigt den vereinfachten Datenpfad und den steuernden Zustandsautomaten.

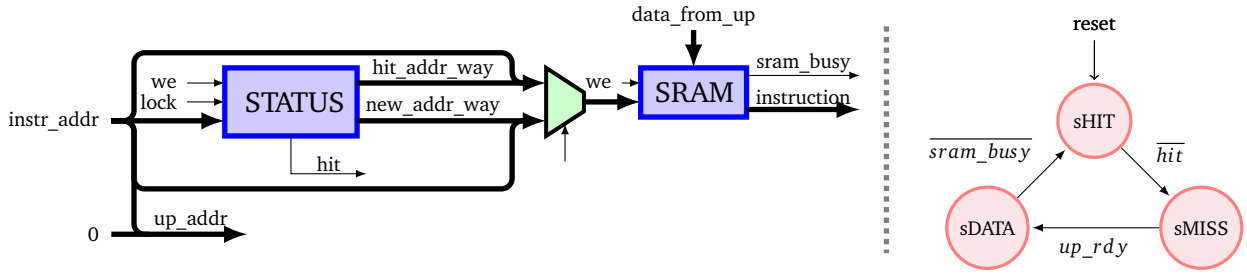


Abbildung 3.11: Der Datenpfad und der Zustandsautomat des Instruktionspufferspeichers der ersten Ebene, vereinfacht dargestellt

Der Pufferspeicher verfügt über zwei Komponenten. Das Statusmodul verwaltet die zugehörigen Informationen, in diesem Fall den jeweiligen *tag* und den *valid*-Bit eines Datenpakets sowie die Ersetzungswarteschlange. Wenn sich die Instruktionsadresse ändert, werden alle *tags* mit dem jeweiligen Teil der Adresse verglichen. Falls es eine Übereinstimmung gibt (und es sollte maximal nur eine pro Adresse geben), wird ein Treffer (*cache hit*) gemeldet. Wenn es nicht der Fall ist, wird ein Fehlzugriff (*cache miss*) signalisiert. Die Kontrolllogik sendet dann eine Anfrage an die höhere Ebene. Wenn die Anfrage angenommen werden kann (das Signal `up_ready` ist gesetzt), werden die Daten in den SRAM-Speicher geschrieben.

Die SRAM-Datenkomponente des Instruktionsspeichers wurde für einen Einsatz in einem FPGA modelliert. Der FPGA-Chip des Herstellers Xilinx stellt einen chipinternen SRAM-Speicher zur Verfügung. Um diesen nutzen zu können, müssen spezielle Modellierungsvorgaben erfüllt sein. Ein Datenpaket der oberen Ebene hatte eine Breite von 256 Bits und entsprach damit 8 Prozessorworten. Die FPGA-spezifische Schnittstelle der SRAM-Speicherkomponente wurde an die Prozessordatenbreite angepasst, um die Lesezugriffszeit möglichst gering zu halten. Dies führte allerdings dazu, dass der Schreibprozess bei einem Fehlzugriff 8 Takte, jeweils einen pro Wort, benötigte.

Der Datenpufferspeicher der ersten Ebene erforderte im Vergleich dazu weit mehr Verwaltungslogik, wie Abbildung 3.12 zeigt.

Neben den auch im Instruktionsspeicher vorhandenen Status- und SRAM-Komponenten muss der Datenspeicher versetzte Datenstrukturen und Schreibzugriffe verwalten können. Beide Aufgaben erforderten eine aufwendige Zusatzlogik, die in weiteren Untermodulen ihren Platz fand. Der Hauptunterschied zur Instruktionskomponente bestand darin, dass auch **Schreibzugriffe vom Prozessor** verarbeitet werden mussten. Insgesamt musste diese Ebene des Pufferspeichers möglichst innerhalb eines Taktes auf die Prozessoranfragen reagieren können. Daher sollte die Schnittstelle zum Prozessor wenige Logikstufen beinhalten. An dieser Stelle ist deutlich, dass jeder Kern eines Mehrkernsystem über mindestens diese Ebene des Pufferspeichers verfügen muss, um die kurze Zugriffszeit erreichen zu können.

Die letzte Schlussfolgerung führte unmittelbar zu einer Entscheidung in Bezug auf die interne Architektur des Datenspeichers. Wenn jeder Kern des Mehrkernsystems über eigene Pufferspeicherebenen verfügte, dann mussten die geschriebenen Daten untereinander synchronisiert werden. Aus diesem Grund wurde die Schreibstrategie *write-through* gewählt, bei der die Daten sowohl in den SRAM-Speicher dieser Ebene als auch (etwas zeitversetzt) auf die höhere Ebene übertragen werden. Dazu müssen nur die jeweiligen Adressen zwischengespeichert werden. Diese Aufgabe übernimmt das Modul `BUFFER`. Die Logik dieses Moduls nimmt neue Adressen nur dann auf, wenn sie sich von den bereits zuletzt gespei-

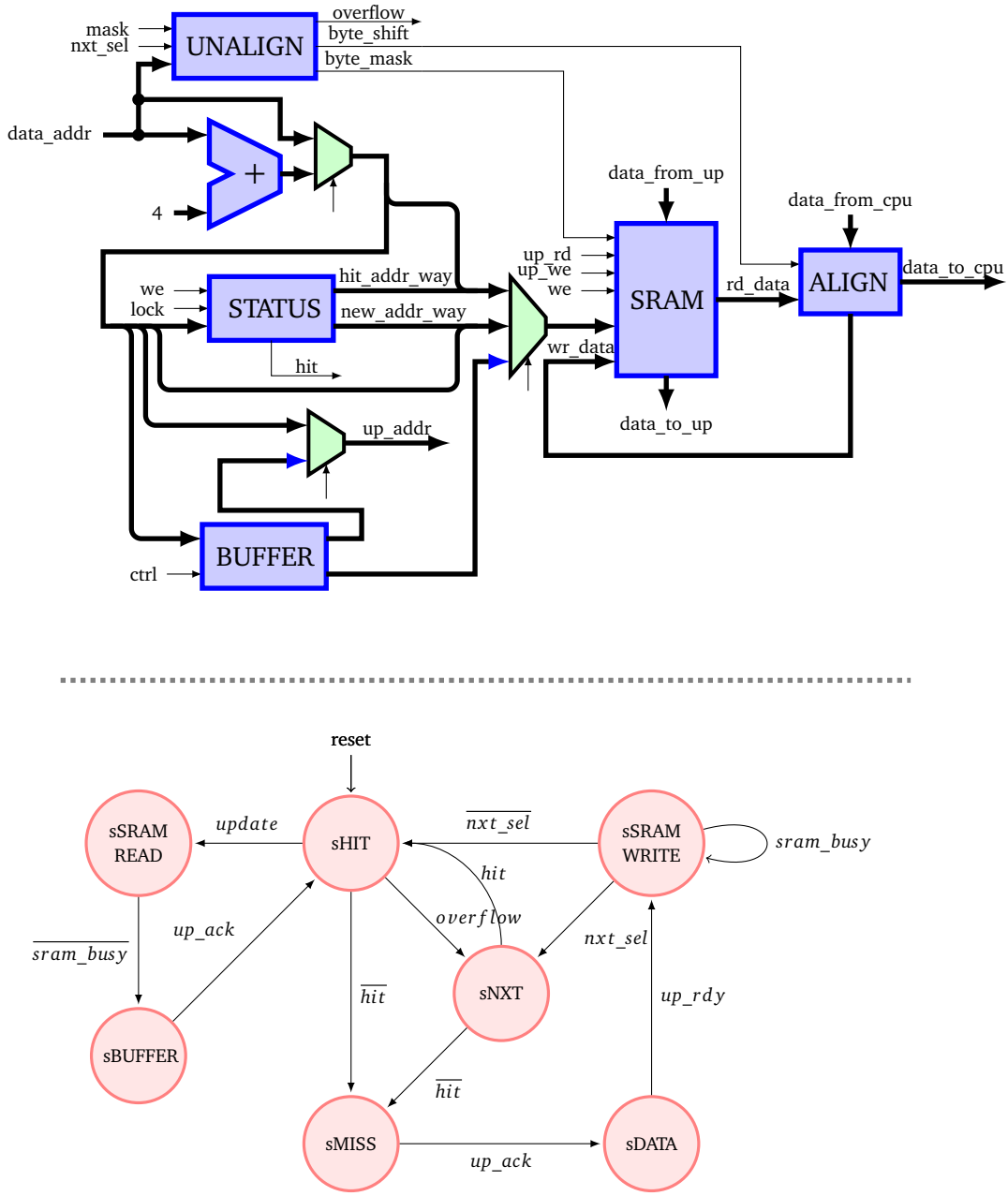


Abbildung 3.12: Der Datenpfad und der Zustandsautomat des Datenpufferspeichers der ersten Ebene, vereinfacht dargestellt

cherten unterscheiden. Dieser Eingriff war nötig, da die Schreibzugriffe aufgrund des Lokalitätsprinzips sehr oft innerhalb eines Datenpaketes stattfanden und somit permanente Übertragungen von kompletten *cache lines* verursachten. Der Adressenvergleich konnte allerdings nur durchgeführt werden, wenn die BUFFER-Komponente die jeweils letzte gültige Adresse behielt. Bei Mehrkernsystemen führte diese Zurückhaltung dazu, dass Datenkonflikte auftraten. Daher sorgt eine zusätzliche Logik dafür, dass sich das BUFFER-Modul nach einer definierten Zeit vollständig leert.

Eine weitere Funktion des Datenspeichers, die eine zusätzliche Logik erforderte, war die Verwaltung von versetzten Daten. Abbildung 3.13 zeigt eine mögliche Beispielkonstellation, bei der die Daten über zwei Pufferspeicheradressen verteilt sind.

Der Pufferspeicher verfügt pro Adresse (*line*) über n Datenpakete (n-assoziativ), die jeweils mit einem *tag* gekennzeichnet sind. Die Datenpakete der ersten Ebene sind 256 Bit breit, enthalten also 8 Worte

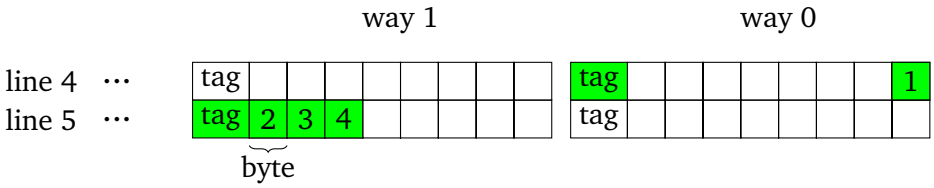


Abbildung 3.13: Position der Daten bei einem Zugriff auf versetzte Daten

je 32 Bit. Da der Speicher byte-adressierbar ist, können Zugriffe auf Adressen stattfinden, die die 4 Bytes aus einem Wort unterschiedlich verteilen. Tabelle 3.3.2 listet alle möglichen Kombinationen und Verteilungen der Bytes.

Tabelle 3.3.2: Kombinationen aus Befehlen und letzten Bits der Adresse, die einen Datenversatz innerhalb der Pufferspeicherzeilen verursachen können.

Befehl	Adresse[1:0]	Byteverhältnis
SH/LH(U)	11	1 / 1
SW/LW	01	3 / 1
	10	2 / 2
	11	1 / 3

Das Problem bei solchen Zugriffen ist, dass die Nummer des Datenpaketes (der *way*) sich unterscheiden kann. Die Daten werden nach der *LRU*-Strategie (zuletzt genutzt, ***L**ast **R**ecently **U**sed*) ersetzt. Eine softwarebasierte Anpassung aller Speicheradressen, die dazu führt, dass diese Adressen nur Vielfache von 32 bilden, war nicht möglich (Näheres dazu im Abschnitt 4.5). Daher überprüft die UNALIGN-Komponente die Datenadressen und signalisiert versetzte Datenwörter. Die Kontrolllogik führt in diesem Fall aus einem weiteren Zustand *sNXT* einen regulären Datenzugriff auf die nächste Pufferspeicheradresse durch. Danach folgt erst der Zugriff auf die angelegte Adresse. Die SRAM-Daten müssen für die CPU entsprechend aus den Bytes von den beiden Zugriffen zusammengestellt werden (Modul ALIGN).

Insgesamt lässt sich festhalten, dass der Datenspeicher eindeutig die wichtigere Komponente in Bezug auf den kritischen Pfad und damit auf die Zugriffszeit darstellt. Tabelle 3.3.3 fasst nochmal alle wichtigen Parameter der Pufferspeicherkomponenten der ersten Ebene zusammen.

Tabelle 3.3.3: Konfiguration der Pufferspeicherkomponenten der Ebene 1

Komponente	Kapazität	Datenpaket	Assoziativität	Schreibstrategie	Verzögerung
Instruktionsspeicher	32 kB	256 Bit	n=32, LRU	-	0
Datenspeicher	32 kB	256 Bit	n=32, LRU	WT & WA	0 bis 1

Die Kapazität und die Aufteilung zwischen der Anzahl der Adressen (*sets*) und der Datenpakete (*ways*) sind bei beiden Komponenten identisch. Lediglich durch die versetzten Datenstrukturen benötigt der Datenspeicher auch im Falle eines doppelten Treffers mindestens einen zusätzlichen Zyklus für die Daten.

Zweite Ebene des Pufferspeichers: alle Daten vereint

Die Konfigurationsparameter der zweiten Ebene des Pufferspeichers sind in Tabelle 3.3.4 dargestellt:

Tabelle 3.3.4: Konfiguration der Pufferspeicherkomponente der Ebene 2

Komponente	Kapazität	Datenpaket	Assoziativität	Schreibstrategie	Verzögerung
Gemeinsamer Speicher	512 kB	1024 Bit	n=32, LRU	WB & WA	3

Auf dieser Ebene verfügt der Pufferspeicher über eine größere Kapazität, die sich in diesem Fall aus 128 Adressen (*sets*) mit je 32 Datenpaketen (*ways*) zusammensetzt. Außerdem können die Instruktionen und Daten gemeinsam platziert werden. Bei einem Mehrkernsystem kann diese Ebene somit als gemeinsamer Speicher agieren. Das macht eine unmittelbare Aktualisierung nach einem Schreibzugriff nicht notwendig. Diese Ebene tauscht Daten über einen Controller mit dem DRAM. Daher erscheint die *write-back*-Schreibstrategie am günstigsten. Bei dieser Strategie werden die Daten nur dann zurückgeschrieben, wenn sie ersetzt werden sollen.

Das Datenpaket auf dieser Ebene ist 1024 Bit breit und enthält 4 *cache lines* der ersten Ebene. Falls die erforderlichen Daten bei einem Zugriff im SRAM enthalten sind, dauert der Datentransfer mindestens 3 Zyklen. Diese Verzögerung ist in erster Linie auf die größere Kapazität zurückzuführen. Des Weiteren verfügt dieser Pufferspeicher über eine zusätzliche Logik, die seinen kompletten Inhalt auf den DRAM übertragen kann (*cache flush*). Aufgrund der *write-back*-Strategie behält dieser Speicher seine Daten auch nachdem eine Applikation die Ausführung bereits beendet hat. Daher muss in diesem Fall der DRAM mithilfe einer zusätzlichen Logik gezwungenermaßen aktualisiert werden. Abbildung 3.14 zeigt den Datenpfad und den Zustandsautomaten der Kontrolllogik des Speichers.

Diese Ebene des Pufferspeichers ist an den DRAM angebunden, sodass alle Fehlzugriffe unmittelbare DRAM-Anfragen verursachen. Die Anzahl und die Häufigkeit dieser Anfragen beeinflussen unmittelbar die Ausführungszeit. Daher muss die Konfiguration dieser Komponente gesondert betrachtet werden. Bei der Modellierung des Pufferspeichers wurde im Allgemeinen darauf geachtet, dass die Implementierung möglichst konfigurierbar bleibt. Die Anpassung der Anzahl der Adressen sowie der Datenpakete pro Adresse war durch eine Anpassung der entsprechenden VHDL-Konstanten möglich. Eine Änderung der Breite eine Datenpaketes war ohne tiefgreifende Veränderungen an der Verarbeitungslogik nicht zu bewerkstelligen, da sowohl die SRAM-Komponente als auch die Übertragungen zwischen den Ebenen eine feste Datenbreite erforderten. Deshalb war eine Variation der Kapazität ohne Weiteres möglich, während die jeweilige *cache line* konstant bleiben musste.

System mit einem Kern

Die vorgestellten Ebenen des Pufferspeichers wurden auf einem System mit nur einem Kern zu einem Pufferspeichersystem verbunden. Dabei waren allerdings noch weitere Komponenten erforderlich (Abbildung 3.15), um den Aufbau einerseits modular zu halten und andererseits an den DRAM anzubinden. Die Modularität beruhte dabei auf zwei Ansätzen:

1. Einheitliches Datenübertragungsprotokoll zwischen allen Ebenen.
2. Keine Änderung der Pufferspeicherkomponenten beim Übergang zum Mehrkernsystem.

Die beiden Komponenten der ersten Ebene konnten bei diesem Ansatz nicht direkt mit der zweiten Ebene verbunden werden, da zwei Datenströme auf nur eine Schnittstelle trafen. Es wurde eine weitere Komponente zwischen den Ebenen geschaltet, die die Anfragen verteilte. Dabei wurden die Zugriffe vom Instruktionsspeicher bevorzugt behandelt, sodass Fehlzugriffe auf den Instruktionsspeicher schneller verarbeitet werden konnten. Für den Datenspeicher der ersten Ebene bedeutet das, dass ein Fehlzugriff im ungünstigsten Fall zu einer doppelten Wartezeit führt.

Das einheitliche Übertragungsprotokoll benötigt neben den Adress- und Datenleitungen noch vier Steuerverbindungen (Abbildung 3.16).

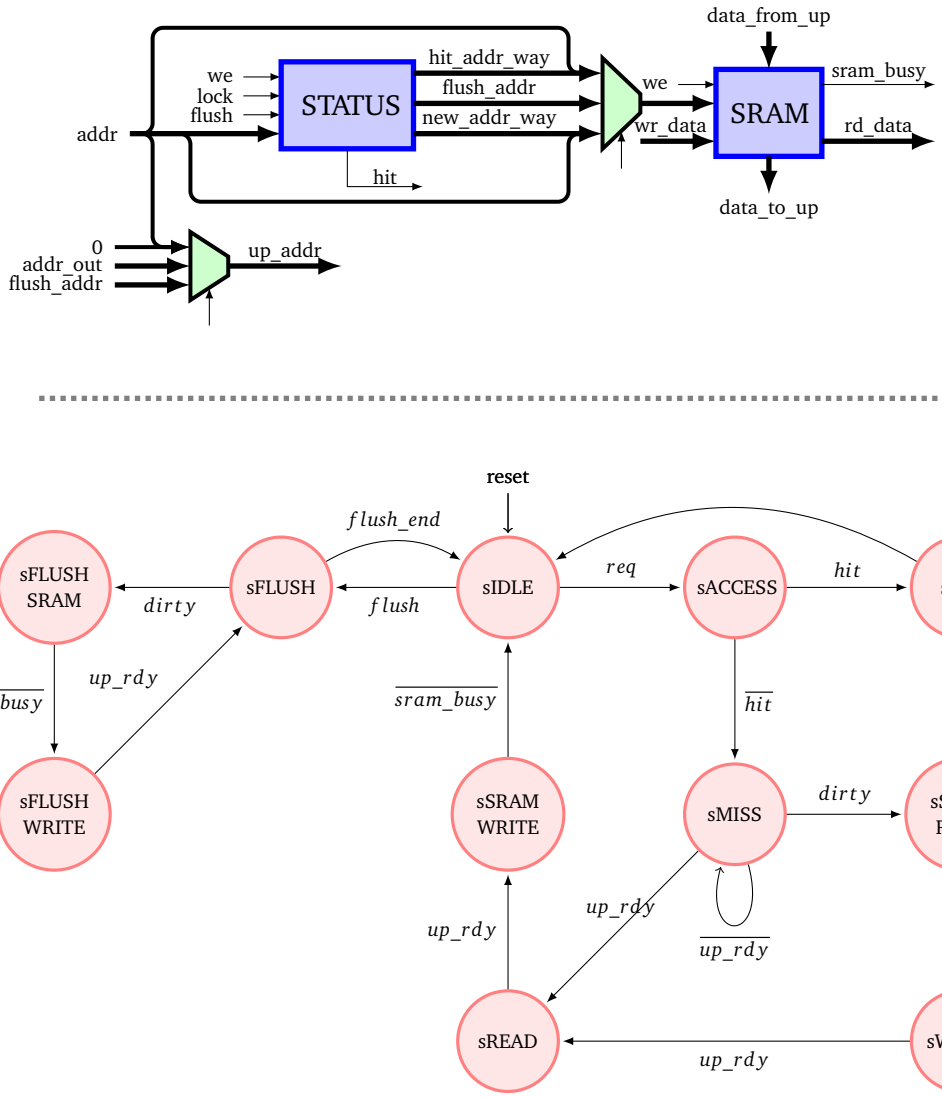


Abbildung 3.14: Der Datenpfad und der Zustandsautomat des Pufferspeichers der zweiten Ebene, vereinfacht dargestellt

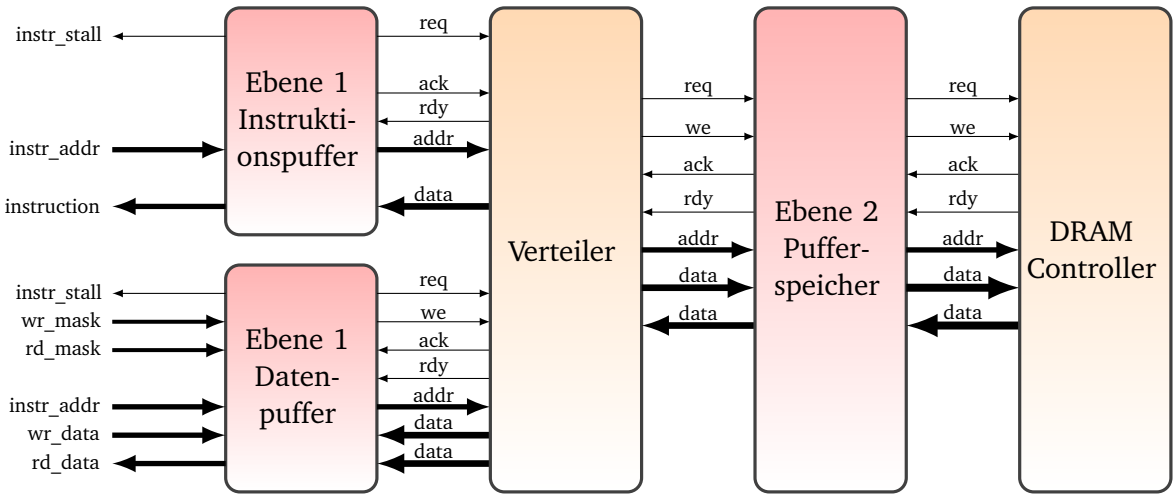


Abbildung 3.15: Aufbau des Pufferspeichers bei einem System mit einem Prozessorkern

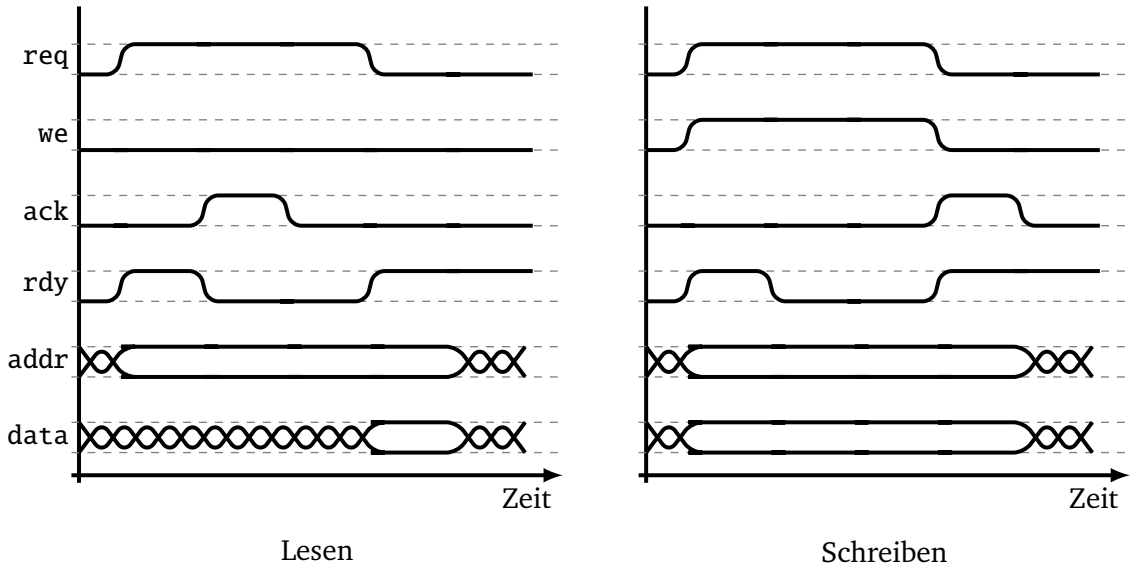


Abbildung 3.16: Übertragungsprotokoll zwischen den einzelnen Modulen des Pufferspeichers

Wenn ein Modul einen Datenzugriff vornimmt, wertet es zunächst das `rdy`-Signal des Gegenübers aus. Solange es nicht gesetzt ist, werden keine Anfragen angenommen. Bei einem Lesezugriff setzt der Empfänger der Daten das `req`-Signal und wartet auf die Daten, die mit einem gesetzten `ack`-Signal übernommen werden können. Ein Schreibzugriff wird durch das `we`-Signal angezeigt und ist mit einer '1' auf dem `ack`-Signal aus Sicht des Senders der Daten abgeschlossen.

Die zweite Ebene des Pufferspeichers konnte mit diesem Protokoll nicht direkt an den DRAM-Controller angebunden werden. Die Datenbreite des *bursts* bei einer Datenübertragung zum DDR2-DRAM betrug zwar 1024 Bit, der Controller konnte aber nur 128 Bit pro Takt annehmen. Daher musste eine zusätzliche Logik die *cache line* der zweiten Ebene entsprechend aufteilen.

Mehrkernsystem

Das Pufferspeichersystem für mehr als einen Kern gestaltet sich nicht nur auf der Gesamtebene komplexer. Abbildung 3.17 zeigt die einzelnen Komponenten beispielsweise für ein Zweikernsystem.

Die Verteilung der Anfragen von der ersten Ebene muss jetzt mit einer viel größeren Anzahl an Schnittstellen zurechtkommen. Der modulare Ansatz erlaubt keine Änderung der zweiten Ebene, sodass die Verteilungskomponente um die entsprechende Logik erweitert wurde. Die Anfragen der jeweiligen Instruktionsspeicher werden weiterhin höher priorisiert. Zusätzlich erfolgt eine Priorisierung zwischen den einzelnen Kernen. Dabei werden die Anfragen der Kerne mit einer kleineren Nummer zuerst weitergeleitet. Somit hat der Instruktionsspeicher des ersten Kerns die höchste Priorität und der Datenspeicher des letzten Kerns die niedrigste. Diese Logik kann im schlimmsten Fall die Kerne mit höherer Nummer für lange Zeit blockieren. Sie eignet sich nur für Lösungen mit kleiner Kernanzahl.

Des Weiteren sind in der Abbildung noch zusätzliche Komponenten der zweiten Ebene angedeutet. Diese Komponenten wurden für weitergehende Untersuchungen verwendet. Ihr Einsatz und ihr Zweck werden in Abschnitt 3.2 näher erläutert. An dieser Stelle sei nur erwähnt, dass auch ein System mit einer separaten Instruktionsspeicherkomponente auf der zweiten Ebene getestet wurde. Die Zugriffszeit dieser Komponenten ist im Vergleich zu der anderen reduziert. In diesem Fall ist ein Verteiler auf der zweiten Ebene mit dem gleichen Priorisierungsprinzip erforderlich.

Außerdem wurde der Datenspeicher der ersten Ebene um eine Datenresetfunktion erweitert, um Aktualisierungen mit der zweiten Ebene zu erzwingen. Wenn beispielsweise Kern 1 Daten an der Adresse X liest, vermerkt seine Datenspeicherkomponente einen Zugriff und aktualisiert entsprechend die Da-

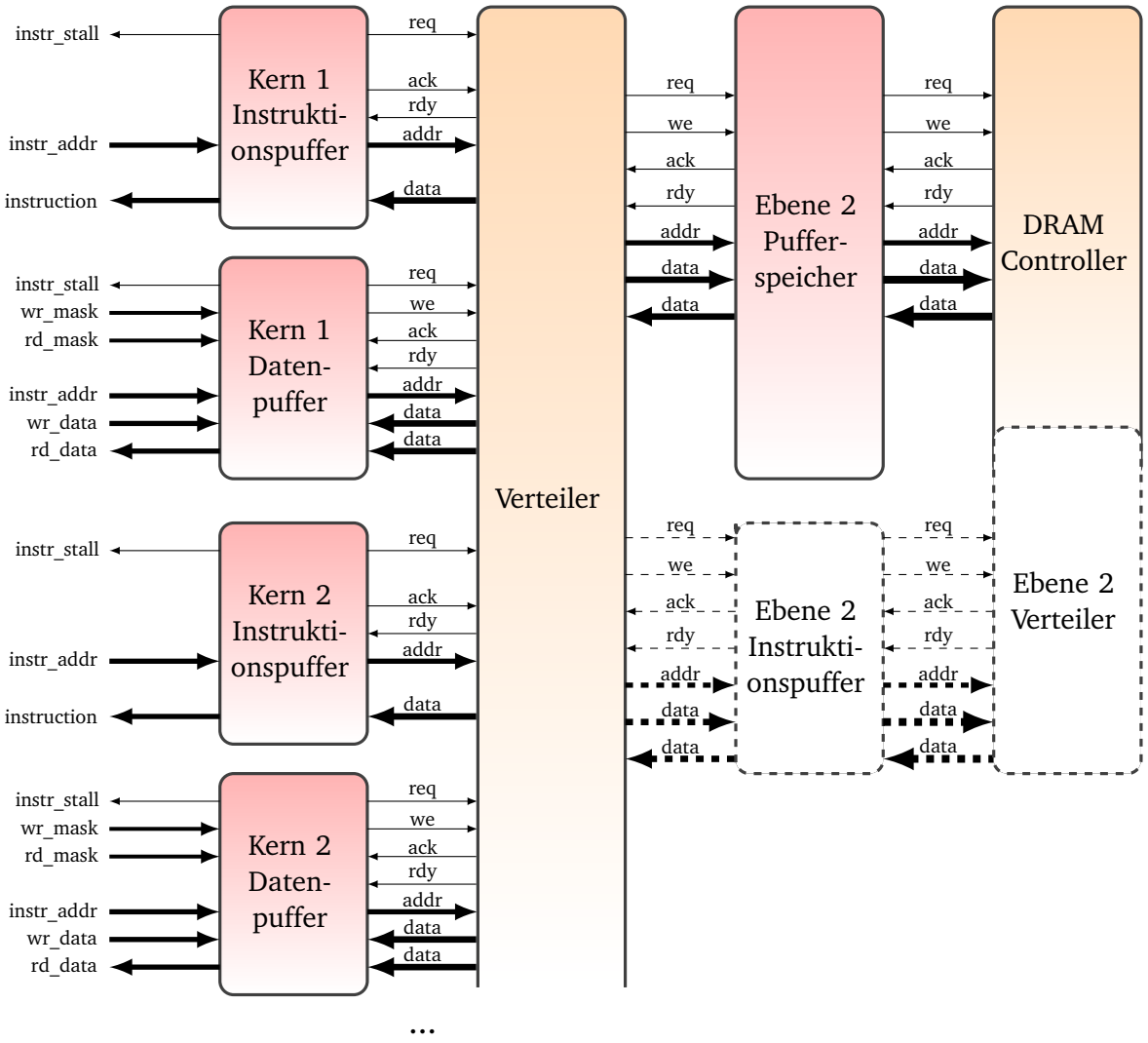


Abbildung 3.17: Aufbau des Pufferspeichers bei einem System mit mehreren Prozessorkernen

tenersetzungsreihenfolge. Wenn der Kern 2 dann an diese Adresse X schreibt, dann werden die neuen Daten nach einer gewissen Zeit zwar auf die zweite Ebene übertragen, für Kern 1 sind sie allerdings nicht zugänglich, da seine Datenkomponente keinen Grund dafür „sieht“, sie zu erneuern. Eine Aktualisierung kann jedoch erzwungen werden, wenn die Daten in Kern 1 als ungültig markiert werden. Genau diese Aufgabe übernimmt die Resetfunktion. Mit ihr können alle Daten des Datenspeichers für ungültig erklärt werden. Dieses Konzept ist riskant und sollte nur selten ausgeführt werden. Diese Pufferspeichermodel eignet sich daher nur für Applikationen, die wenig bis gar keine Datenabhängigkeiten haben.

Insgesamt lässt sich feststellen, dass der Pufferspeicher für ein Mehrkernsystem wesentlich höhere Anforderungen in Bezug auf die Datenkonsistenz erfüllen muss. Die Implementierung erhöht die Komplexität der Logik, sodass mehr Fehlerquellen entstehen.

3.3.3 3D-DRAM-Modell

Unter allen Komponenten für die Testumgebung war die Modellierung eines gestapelten DRAM die kritischste. Die komplette Untersuchung dieser Arbeit hing von einer Lösung dieser Aufgabe ab. Dabei offenbarte sich ein Dilemma: Wie soll eine Komponente möglichst umfassend modelliert werden, wenn ihre physikalische Realisierung noch gar nicht existiert?

Die Abstraktion eines Objekts kann sowohl die statische (Struktur) als auch die dynamische (Verhalten) Natur dieses Objekts wiedergeben. Eine Strukturbeschreibung lässt Randbedingungen erkennen, während ein Verhaltensmodell die Zeitdimension berücksichtigt. Das DRAM-Referenzmodell von Micron verzichtet beispielsweise weitgehend auf eine explizite Strukturbeschreibung und kapselt sie in Funktionen ein. Diese Funktionen lassen zwar auf bestimmte Komponenten zurückschließen, ihre Hauptaufgabe besteht allerdings darin, die jeweiligen Verzögerungszeiten möglichst genau und gleichzeitig parametrisierbar zu modellieren. Wenn ein realer Baustein aufgrund dieses Modells entworfen werden soll, ist ein Strukturkonzept unverzichtbar. Es kommt also auf die jeweilige Aufgabe an, die mithilfe des Modells bearbeitet wird.

Die Fragestellungen dieser Arbeit erfordern sowohl ein Verhaltens- (Frage 3) als auch ein Strukturmodell (Frage 2). Es sollte also möglich sein, bestimmte Strukturanordnungen mithilfe eines Modells testen zu können und gleichzeitig eine mögliche Realisierung zu berücksichtigen. Die Stapeltechnik erlaubt dagegen eine Fülle an möglichen Designansätzen, wie die Auflistung der aktuellen Forschungsarbeiten in Abschnitt 2.2.2 zeigt. Es ist auch davon auszugehen, dass damit keineswegs alle möglichen Herangehensweisen untersucht wurden. Letzten Endes muss die technische Realisierung zeigen, inwieweit diese Ansätze den Anforderungen der Industrie gerecht werden können. Die Implementierung eines kompletten DRAM - beginnend mit der Konzeption, der Modularisierung, dem Schaltungsentwurf, dem Layout, der Herstellung und der Inbetriebnahme - hätte allerdings den Rahmen dieser Arbeit ohne zusätzliche Hilfestellung gesprengt. Es mussten also gewisse Annahmen in Bezug auf die Realisierung getroffen werden. Diese Annahmen engten den Untersuchungsraum zwar ein, lieferten aber einen ersten Anhaltspunkt.

Die wichtigste Annahme im Hinblick auf eine mögliche Realisierung eines 3D-DRAM war folgende:

Annahme 1

Der prinzipielle Aufbau einer Bank entspricht bei einem gestapelten DRAM der konventionellen Technik.

Ausgehend von dieser Annahme konnte die DRAM-Bank als kleinste Struktur angesehen werden, die nur in Bezug auf ihr Verhalten modelliert werden muss. Auch die entsprechenden Zugriffssignale konnten so wiederverwendet werden. Abbildung 3.18 zeigt die Signalverläufe und die dazugehörigen Zeiten.

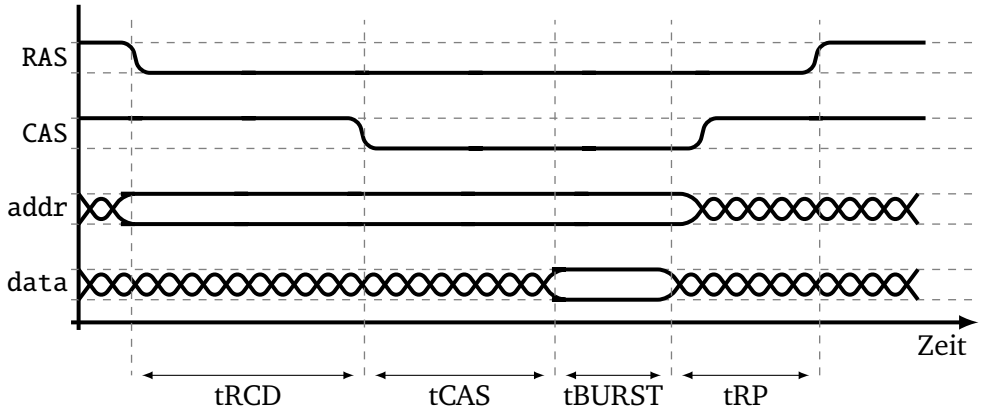


Abbildung 3.18: Signalverläufe und Zeitkonstanten bei einem Zugriff auf eine Bank

Fragen hinsichtlich der Anzahl und der Anordnung solcher Bänke, darüber hinaus hinsichtlich der Anzahl der gestapelten Schichten, sind Strukturfragen. Das Literaturstudium (Abschnitt 2.2.2) zeigte, dass

sowohl [Woo et al., 2010] als auch [Weis et al., 2013] in ihren Berechnungen unabhängig voneinander auf die gleiche Gesamtstruktur eines 3D-DRAM kommen, die sich als optimal in Bezug auf die Zell- und Energieeffizienz erweist. Die Herangehensweise der Forscher, insbesondere bei [Weis et al., 2013], beruht auf einem detaillierten Speichermodell. Ihre Ergebnisse wurden daher als Ausgangspunkt für das Modell bei dieser Arbeit genommen. Abbildung 3.19 zeigt den logischen Aufbau.

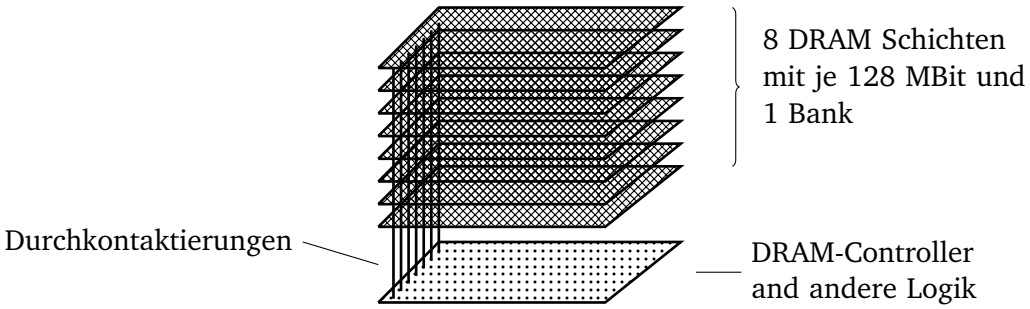


Abbildung 3.19: Strukturmodell eines 3D-DRAM: die Gesamtkapazität beträgt 1 Gigabit (2^{30} Bit).

Der Speicher ist aus acht Schichten aufgebaut. Die Durchkontaktierungen sind räumlich zusammengefasst. Die tatsächliche physikalische Platzierung der Logikschicht und eventuell vorhandener weiterer Schichten bleibt in diesem Modell frei. Die Anzahl der Bänke dagegen ist fest auf eine Bank pro Schicht festgelegt, da diese Anordnung ein Optimum darstellt [Weis et al., 2013]. Dieses Modell wurde für diese Arbeit in SystemVerilog implementiert. Der Einsatz dieses Modells dient als Bezugsgröße für die Hypothesenüberprüfungen.

Die Stapeltechnik erlaubt es, die Datenbreite zu erweitern und gleichzeitig die Übertragungsfrequenz zu steigern. Um theoretische Grenzen feststellen zu können, wurde bei dieser Arbeit folgende Annahme getroffen:

Annahme 2

Es ist möglich, innerhalb einer 3D-Schicht nur die asynchrone Kernkomponente der DRAM-Bank zu realisieren.

Die komplette Peripherielogik inklusive der Synchronisation wurde ausgelagert, sodass die Schnittstelle des 3D-DRAM nur noch die Grundsignale der DRAM-Technik benötigt (Abbildung 3.20).

Jeder Datenvektor steht für eine separate Verbindung mit einer Schicht, die allerdings über gemeinsame Signale angesteuert wird. Diese 1024 Bit breite Datenanbindung ist den Untersuchungen von [Woo et al., 2010] entnommen. Diese Breite führte zwar nicht bei jeder Applikation zu der geringsten Fehlzugriffsrate, im Vergleich zu anderen Datenbreiten zeigte dieser Wert aber in der Regel ein lokales Minimum.

Controller

Die Einbindung der asynchronen DRAM-Bänke erfordert eine zusätzliche Logik, die die Einhaltung von Zugriffszeiten übernimmt und zudem für die Synchronisierung der Daten zuständig ist. Damit übersteigt der Funktionsumfang des verwendeten Controllers den von konventionellen Lösungen, die einen nach außen synchron agierenden DRAM-Chip ansteuern. Der Entwurf eines DRAM-Controllers umfasst ohnehin eine Fülle von Aufgaben, sodass diese Komponenten der Testumgebung in dieser Arbeit möglichst auf ein Minimum zu reduzieren versucht wurde.

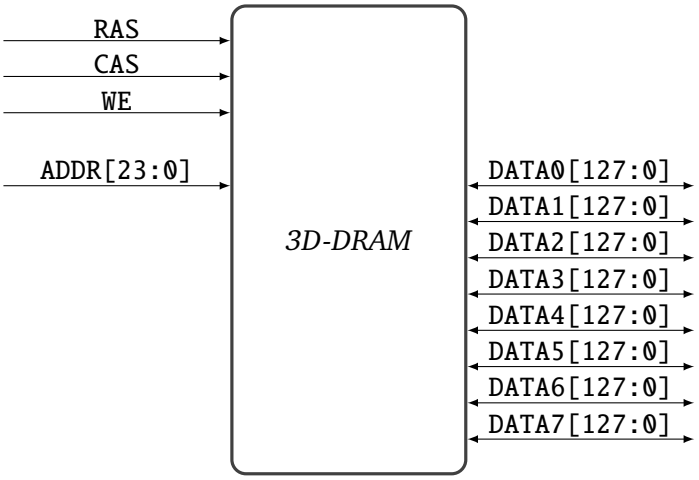


Abbildung 3.20: Außenschnittstelle des 3D-DRAM

Die Komponente wurde in der Hardwarebeschreibungssprache SystemVerilog implementiert, da diese Sprache über Konzepte verfügt, die Zeitverzögerungen ohne großen Aufwand implementieren lassen. Die dynamische Natur eines DRAM wird durch eine Aktualisierungslogik berücksichtigt. Alle 32 Millisekunden simulierter Zeit wird der Speicher für eine einstellbare Zeitkonstante blockiert. Währenddessen keine Datenzugriffe angenommen werden.

Außerdem wurde eine optional zuschaltbare Optimierungsstufe eingefügt. Die Schreibstrategie des Pufferspeichers auf der letzte Ebene führt dazu, dass die Daten nur zurückgeschrieben werden, wenn sie ersetzt werden sollen. Auf einen Schreibzugriff folgt also unmittelbar ein Lesezugriff. Die Optimierung besteht darin, die Reihenfolge der Ausführung dieser beiden Befehle zu vertauschen. Die Daten des Schreibzugriffs werden temporär gespeichert. Dem Pufferspeicher wird signalisiert, dass der Schreibprozessor bereits abgeschlossen ist, ohne auf DRAM-Bänke zuzugreifen. Der folgende Leseprozess wird regulär ausgeführt. Während des Rest des Systems mit den neuen Daten beschäftigt ist, wird der Schreibvorgang eingeleitet. Durch diese Maßnahme kann die Fehlzugriffsverzögerung bis auf die Hälfte (abhängig von der Taktfrequenz) reduziert werden.

Die Applikationsschnittstelle des DRAM-Controllers (Abbildung 3.21) nutzt das gleiche Übertragungsprotokoll wie das Pufferspeichersystem (Abbildung 3.16).

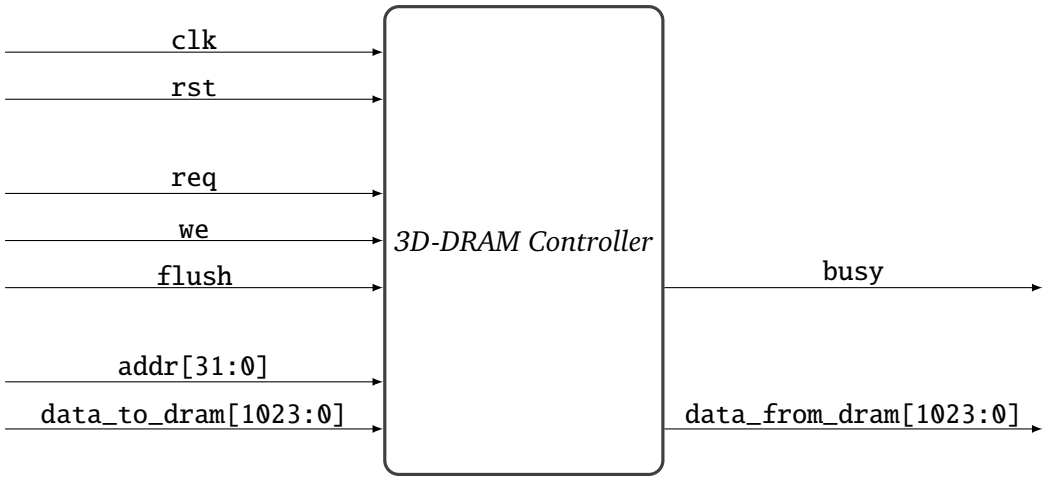


Abbildung 3.21: Applikationsschnittstelle des 3D-DRAM-Controllers

Außerdem zeigt das Signal `flush` an, dass ein entsprechender Datenübertragungsprozess des Pufferspeichers gerade stattfindet und somit nur Schreibbefehle zu erwarten sind. Dieses Signal ist nur bei

eingestellter Optimierungsstufe relevant, da die Optimierungslogik nach jedem Schreibzugriff einen Lesezugriff erwartet. Eine Verzögerung durch die Logik des DRAM-Controllers wurde für Simulationen nicht modelliert, um nur den DRAM-Anteil betrachten zu können.

Parametrisierung

Tabelle 3.3.5: Parametrisierbare Variablen des 3D-DRAM-Models und ihre Anfangswerte

Variable	Initialwert
tRCD	13 ns
tCAS	13 ns
tBURST	1 ns
tRP	12 ns
tREFRESH	40000 ns

Eine wichtige Funktion bei der Modellierung von Bausteinen, die noch keiner physikalischen Realisierung entsprechen, ist die Parametrisierbarkeit. In Bezug auf das vorgestellte Modell konnte diese Funktion im Wesentlichen nur auf die Verzögerungszeiten angewendet werden. Die Kapazität war weitgehend festgelegt. Konzepte eines virtuellen Speichers, die bei mangelnder Hauptspeichergröße greifen würden, waren nicht vorgesehen. Tabelle 3.3.3 listet die wichtigsten parametrisierbaren Konstanten und die dazugehörigen Anfangswerte auf.

Die Zeiten betreffen in erster Linie den Zugriffsprozess einer DRAM-Bank, der kleinsten Strukturgröße im Modell. Da in dieser Arbeit davon ausgegangen wird, dass diese Bänke auf Lösungen von konventionellen Bausteinen basieren, sind die Anfangswerte auch einem realen DRAM-Chip entnommen. Es ist ein DDR3-DRAM des Herstellers Micron, der im Modus *DDR3-2133* betrieben wird [Micron Technology, Inc., 2006].

Pufferspeicherfunktionalität: latenzoptimierte Schicht

Die Hypothesen dieser Arbeit gehen von bestimmten Annahmen bezüglich der Möglichkeiten der Stapeltechnik und des DRAM aus:

Annahme 3

Die Stapeltechnik ermöglicht kürzere Zugriffszeiten auf den DRAM.

Annahme 4

Die Stapeltechnik ermöglicht eine Implementierung von Pufferspeicherfunktionalität im DRAM.

Die Annahme 3 geht aus der Tatsache hervor, dass die Schnittstelle zwischen einem gestapelten DRAM und der restlichen Logik im Vergleich zu konventioneller Technik mit einer höheren Taktfrequenz betrieben werden kann. Kürzere Leitungslängen und abschätzbare Leitungsimpedanzen erlauben diese Frequenzsteigerung. Die tiefer gehenden Annahmen dieser Arbeit behaupten, dass nur noch Pufferspeicherfunktionalität und nichts anderes zu weiterer Laufzeitverringerung aufgrund der Stapeltechnik führen kann (Hypothesen 2 und 3). Diese Funktionalität lässt sich wie folgt formulieren:

Pufferspeicherfunktionalität bedeutet, dass häufig genutzte Instruktionen und Datenstrukturen mit einer geringeren Latenz zur Verfügung gestellt werden als die restlichen Daten.

Die Überprüfung der Hypothesen erfordert eine geänderte Architektur des gestapelten DRAM, sodass manche Daten mit einer noch geringeren Latenz übertragen werden können. Die direkte Nutzung der DRAM-Zellen als Pufferspeicher wurde bereits bei [Lee et al., 2015] vorgestellt. Die Herausforderung beim Entwurf des Pufferspeichers besteht darin, die Größe der Datenpakete und ihre Anzahl pro Adresse festzulegen. So zutreffend das Lokalitätsprinzip für die meisten Applikationen sein mag, so wenig ist es dazu in der Lage, ein konkretes Verhalten vorherzusagen. Da aber die Konfiguration des Pufferspeichers nach der Herstellung nicht mehr verändert werden kann, hängt seine Wirksamkeit und der Ressourceneinsatz davon ab, wie gut er die Lokalitätseigenschaften der jeweiligen Applikation trifft. Daher ist der Hauptvorschlag dieser Arbeit, die Pufferspeicherfunktionalität teilweise in die Software zu verlagern. Die dazugehörige Hardwarearchitektur des 3D-DRAM ist in Abbildung 3.22 dargestellt.

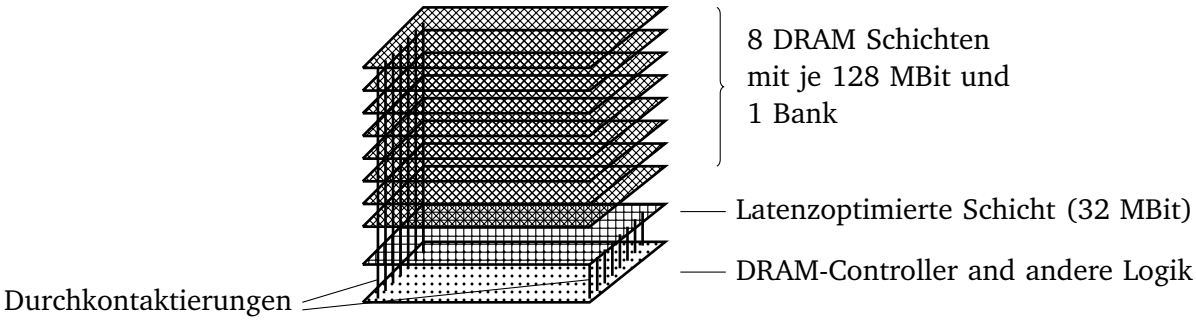


Abbildung 3.22: Strukturmodell des 3D-DRAM mit einer zusätzlichen latenzoptimierten Schicht.

Im Unterschied zum Referenzmodell in Abbildung 3.19 verfügt dieser Speicher über eine latenzoptimierte Schicht, die die Zugriffszeit auf die Hälfte reduziert. Diese Reduktion ist, wie in Abschnitt 2.1.3 hergeleitet wurde, im Wesentlichen nur durch Kapazitätsreduktion zu erreichen. Daher wird für die Untersuchungen angenommen, dass die Kapazität durch die zusätzlichen Leitungen auf ein Viertel der übrigen Schichten sinkt. Diese Annahme ist vermutlich zu pessimistisch für ein reales System und bildet daher eine Untergrenze für mögliche Verbesserungen.

Im Unterschied zum Pufferspeicher ist die latenzoptimierte Schicht Teil des Adressraums des Hauptspeichers. Sie behält nicht durch eine hardwareimplementierte Ersetzungslogik häufig genutzte Daten. Die Schicht agiert genauso wie die üblichen Bänke des DRAM und ist durch Speicherzugriffe innerhalb einer definierten Adressspanne zu erreichen, wie in Abbildung 3.23 veranschaulicht wird.

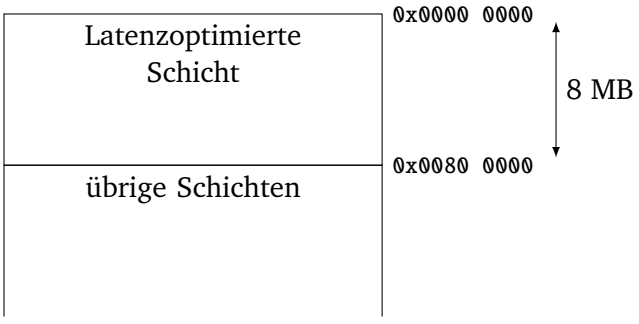


Abbildung 3.23: Adressraum der latenzoptimierten Schicht.

Erst die Stapeltechnik macht diese Architektur möglich, da die spezielle Schicht zusätzliche Durchkontaktierungen erfordert. Der Controller kommuniziert dann abhängig von der aktuellen Adresse entweder über die regulären Datenleitungen oder mithilfe einer zusätzlichen Verdrahtung.

Die Verlagerung der Pufferspeicherkapazität auf die Software besteht darin, dass die häufig genutzten Instruktionen und Datenstrukturen direkt in der latenzoptimierten Schicht platziert werden. Dies erfordert zwar Anpassungen auf der Softwareseite, diese Anpassungen sind dann aber auf die spezielle Applikationen abgestimmt und durch Softwareänderung jederzeit modifizierbar.

Abschließend ist noch anzumerken, dass die physikalische Realisierung einer solchen Architektur zwei Herausforderungen zu bewältigen hätte. Erstens erfordert die Umleitung des Datentransfers zwischen den regulären Schichten und der latenzoptimierten Schicht eine zusätzliche Logik. Diese Logik kann die Latenz unnötigerweise vergrößern, während das Hauptziel gerade deren Verringerung ist. Der zweite Aspekt betrifft die allgemeine Herausforderung der Stapeltechnik: die Verwaltung der Abwärme. Für diese Architektur trifft diese Herausforderung in besonderem Maße zu, da die latenzoptimierte Schicht vom Prinzip her häufig genutzt werden sollte. Es ist somit von einer entsprechenden Wärmeentwicklung auszugehen.

4 Testumgebung und -applikationen

4.1 *3DMemory* – ein Analysewerkzeug

Neben der direkten Messung der Laufzeit war eine tiefer gehende Analyse der Ausführung einer Applikation erforderlich. Die vorgeschlagene Architektur des 3D-DRAM (Abschnitt 3.3.3) verlagerte die Pufferspeicherfunktionalität in die Software, sodass die häufig genutzten Datenstrukturen im Voraus erkannt werden mussten. Außerdem benötigt Hypothese 3 einen umfassenden Ansatz. Daher wurde die größtmögliche Erfassungsmethode angewandt: die Aufzeichnung sämtlicher Speicherzugriffe. Für die Analyse wurde eine gesonderte Softwarelösung implementiert: *3DMemory*.

4.1.1 Gewinnung der Speichernutzungsdaten

Um Aussagen über die Leistungsfähigkeit der gegebenen Speicherkonfiguration treffen zu können, kann ein Blick auf die konkreten Speicherzugriffsdaten neue Erkenntnisse liefern. Insbesondere im Bereich der Pufferspeicherentwicklung wird dieser Ansatz verfolgt. Dabei kann eine Schätzung der Fehlzugriffsrate die nötige Grundlage bilden [Berg und Hagersten, 2004]. Alternativ kann wie schon bei [Hennessy und Patterson, 2012] oder bei [Ismail et al., 2013] eine genaue Erhebung aller Zugriffe (*trace-driven approach*) ausgewertet werden. Der erste Ansatz kann bereits aus der gegebenen Applikation Ergebnisse liefern, ohne dass diese Applikation ausgeführt wird. Die Erhebung aller Zugriffe produziert dagegen eine große Menge an Daten, deren Gewinnung und Analyse erheblich mehr Zeit in Anspruch nimmt als eine Schätzung. Dafür liefern genaue Daten entsprechend detaillierte Analyseergebnisse.

Zusammenfassend lässt sich also folgern, dass es, wie so oft, die Randbedingungen sind, die über die Analyseform entscheiden. Wenn genügend Zeit- und Ausstattungsressourcen vorhanden sind, lassen sich auch die Speicherzugriffsdaten im entsprechenden Umfang erfassen. Dagegen kann eine Schätzung bei einem akzeptierten Fehler als Randbedingung durchaus belastbare Ergebnisse bei vergleichsweise geringem Aufwand liefern. In dieser Arbeit wurde der *trace-driven*-Ansatz verfolgt.

4.1.2 Beschreibung des *3DMemory*

Entwurf

Die Ergebnisse einer Analyse können je nach Fragestellung unterschiedliche Aspekte beleuchten. Daher wurde eigens für diese Arbeit eine Softwarelösung entworfen, die es erlaubt, dieselben Rohdaten einer Speichernutzung mit unterschiedlichen Analysemethoden zu durchleuchten. Abbildung 4.1 zeigt den Strukturaufbau des *3DMemory*.

Der Entwurf sieht drei Hauptkomponenten vor. Die Komponente `Import` ist für die Aufnahme der Rohdaten sowie Systemkonfiguration und Technologieparameter zuständig. Die `Analyse`-Komponente liefert eine Basiserfassung der Speicherzugriffe, wie die Verteilung auf einzelne Speicherabschnitte (wie Instruktionsbereich, Dateneingabenbereich, Stack usw.) und - bei einem Mehrkernsystem - zusätzlich auf die jeweiligen Kerne. Die Hauptfunktion dieser Komponenten ist allerdings die Bereitstellung einer Schnittstelle für eine tiefer gehende Analyse. Diese externe Komponente kann mehrere Analyseansätze

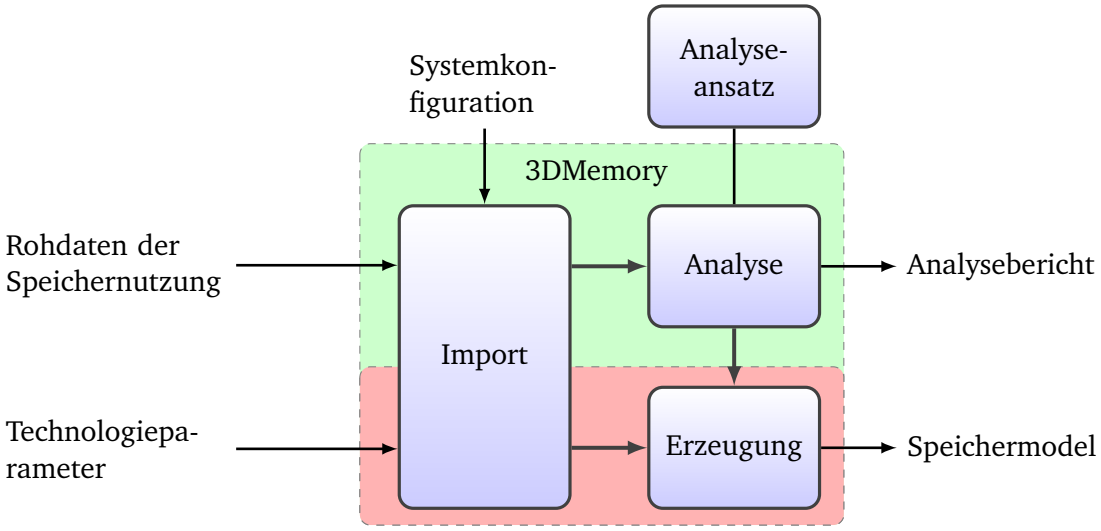


Abbildung 4.1: Komponenten des *3DMemory* und der Datenfluss [Schoenberger und Hofmann, 2014a]

enthalten und über die Systemkonfiguration gesteuert werden. Im Anschluss an die Analyse wird ein Bericht erstellt, der sowohl als Textdatei für den Entwickler zur Verfügung steht, als auch intern für eine Erzeugung eines entsprechenden Speichermodells weiterverwendet werden kann.

Das `Erzeuger`-Modul berücksichtigt importierte Technologieparameter und liefert ein Speichermodell, dass das Verhalten nachbildet auch Strukturangaben für eine Synthese enthält. Dieser Teil des *3DMemory* kam im Gegensatz zum Analysepfad (beschrieben im vorigen Absatz) nur bis zur Konzeptphase, da die Ergebnisse der Analysen bereits ausreichend Informationen lieferten, um die Untersuchungen dieser Arbeit fertigstellen zu können. Dieser Teil der Software wird daher im weiteren Verlauf nicht mehr berücksichtigt.

Systemkonfigurationsangaben

Die Rohdaten der Speicherzugriffe können nur dann zugeordnet werden, wenn Grundinformationen über das untersuchte System vorliegen. Ausschnitt 4.1.2 zeigt ein Beispiel für Angaben, die für eine Zuordnung mindestens benötigt werden.

4.1.2 Basiskonfigurationsangaben für eine Analyse der Speichernutzungsdaten

```
// ########## GENERAL INFORMATION #################
#core_number            =2            // number of cores
#dram_size              =18882560     // DRAM size

// ########## MEMORY ############
//  ADDRESSES
#addr +instr            =0            // instruction
#addr +idata            =8388608      // input data
#addr +odata            =9240576      // output data
#addr +stack            =10092544     // stack
#addr +heap             =10616832     // heap

// LENGTHS
#length +instr          =204508       // instruction
#length +heap           =16777216     // heap
```

Die Schlüsselwörter für die Hauptinformation werden durch das reservierte Zeichen # eingeleitet. Nach einem Gleichheitszeichen = folgt der Wert. Die Hauptinformationsangaben lassen sich mithilfe des Pluszeichens + noch weiter aufgliedert. *3DMemory* wurde in der Programmiersprache C++ realisiert, was sich an der Kennzeichnung der Kommentare (Zeichen //) widerspiegelt.

Die Information über die Anzahl der Kerne wird direkt dazu verwendet, um den Import der Rohdaten der Speicherzugriffe zu steuern. Die Adress- und Größenangaben werden erst bei der Basisanalyse verwendet. Des Weiteren kann die Systemkonfigurationsdatei Angaben über die Art der tiefer gehenden Analyse enthalten, wie Ausschnitt 4.1.2 zeigt.

4.1.2 Beispiel für eine Auswahl der tiefer genden Analyse

```
#analysis +dependency   =0          // data dependency analysis
#analysis +locality     =1          // frequently used functions and data structures
```

In dem gezeigten Beispiel wird die Lokalität der Daten untersucht. Eine Schaltung der dependency-Analyse durch eine 1 würde dagegen gemeinsam genutzte Speicherbereiche bei einem Mehrkernsystem liefern.

Rohdaten der Speichernutzung

Die Erfassung aller Speicherzugriffe liefert Daten, die eine exakte Analyse der Speichernutzung erlauben. Der Nachteil dieser Methode liegt in ihrem Umfang. Selbst kurze Laufzeiten können große Datenmengen produzieren. Diese Datenmengen müssen verwaltet und auch analysiert werden können. Ein weiterer Ansatz fasst die Erfassung und Analyse zu einem Vorgang während der gesamten Ausführung zusammen. Die entscheidende Frage stellt sich dabei immer wieder: Welche Datenmenge ist für die angestrebte Analyse ausreichend?

Die Gewinnung dieser Rohdaten ist nicht Teil des *3DMemory*. Daher wurde in der ersten Version der Software die einfach zu portierende ASCII-kodierte Textform für Daten genutzt. Somit konnten sowohl die Gewinnung aus der Simulation als auch die Erfassung durch das `Import`-Modul ohne großen Aufwand realisiert werden. Die notwendigen Informationen konnten auf ein Datenpaar pro Zugriff reduziert werden. Das erste Teil des Paars kodiert den Vorgang. Die entsprechenden Zeichen sind der Tabelle 4.1.2 zu entnehmen.

Tabelle 4.1: Reservierte Zeichen für bestimmte Speicherprozesse während der Ausführung

Zeichen	Vorgang
l	Lesen
s	Schreiben
n	keine Aktion
m	Marker

Es können also auch Taktvorgänge gekennzeichnet werden, die keine Speicherzugriffe auslösen. Die Erfassung solcher Takte erlaubt Rückschlüsse darauf, welchen Anteil an der Ausführungszeit der Speicher mitbestimmt. Des Weiteren kann der „Marker“ für Übermittlung bestimmter Informationen dienen, um beispielsweise nur einen Ausschnitt der Ausführung zu analysieren. In diesem Fall schaltet das erste Auftauchen des „Markers“ den entsprechenden Prozess ein. Das zweite Auftauchen schaltet ihn wieder aus.

Das zweite Teil des Datenpaars beinhaltet die jeweilige Speicheradresse. Die übertragenen Daten werden nicht mit erfasst, da sie für die Analyse in dieser Arbeit nicht relevant waren.

4.2 Betriebssystemfunktionalität

Die komplette Software, die auf einem Datenverarbeitungssystem ausgeführt wird, muss über bestimmte Informationen über die Hardware verfügen. Dabei ist es vorteilhaft und aus diversen Gründen gar notwendig, den Zugriff auf diese Informationen nur einer bestimmten Abstraktionsschicht zuzulassen. Der Übergang zu dieser Schicht kann allerdings fließend sein und eine Applikation kann über optionale Module verfügen, die für bestimmte Hardwarekonfigurationen direkte Assemblerbefehlsfolgen bereitstellen. Dagegen kann die Speicherverwaltung beispielsweise weiterhin über Funktionsaufrufe externer Bibliotheken realisiert werden [Tanenbaum, 2009].

Für die Testapplikationen dieser Arbeit musste die Abstraktionsschicht neu implementiert werden. Die Schicht besteht aus einer Reihe von Standardfunktionen der C-Bibliothek, wobei nur die notwendigen Abläufe implementiert wurden. Diese Neuimplementierungen stellten zusätzliche Fehlerquellen dar. Daher bedurfte die Entscheidung darüber, welche Prozesse tatsächlich notwendig sind, einer Analyse der Applikation. Abbildung 4.2 veranschaulicht den Sachverhalt.

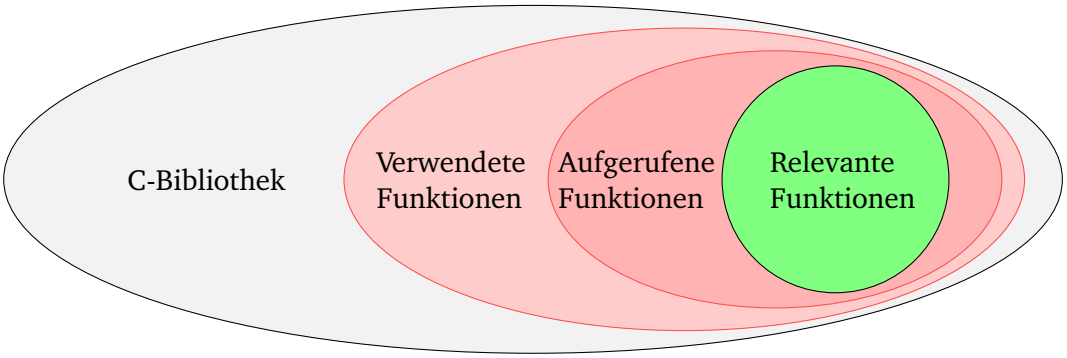


Abbildung 4.2: Untermenge der implementierten Betriebssystemfunktionen

Die Abbildung verdeutlicht, dass selbst aufgerufene Funktionen nicht unbedingt für die Ausführung der Applikation relevant waren. Die Menge der verwendeten Funktionen umfasst alle Aufrufe von externen Bibliotheken im Quellkode. Diese wurden allerdings nicht ausgeführt. Die tatsächlich implementierten relevanten Funktionen lassen sich wie folgt in mehrere Kategorien einteilen:

4.2.1 Speicherverwaltung

Eine der wesentlichen Aufgaben, die ein Betriebssystem übernimmt, ist die Speicherverwaltung. Dabei kann der Speicher grob in fünf Bereiche aufgeteilt werden, die jeweils unterschiedliche Daten beinhalten:

- **Instruktionen**: Alle Instruktionen sowie Compilerkonstanten und globale Variablen
- **Eingangsdaten**: Daten, die für die Verarbeitung eingelesen werden.
- **Ausgangsdaten**: Daten, die nach ihrer Verarbeitung geschrieben werden.
- **Stack**: Speicherbereich, der während der Ausführung einer Applikation für Zwischenergebnisse und sonstige temporär benutzte Daten gebraucht wird.
- **Heap**: In diesem Speicherbereich werden dynamisch erzeugte Datenstrukturen gespeichert.

Diese Aufteilung stellt eine klassische Anordnung dar und kann je nach Bedarf durch Zusammenfassung der Funktionalitäten um einen oder mehrere Bereiche reduziert werden. Die Testapplikationen dieser Arbeit verwenden dagegen den kompletten Satz. Insbesondere die dynamische Verwendung des

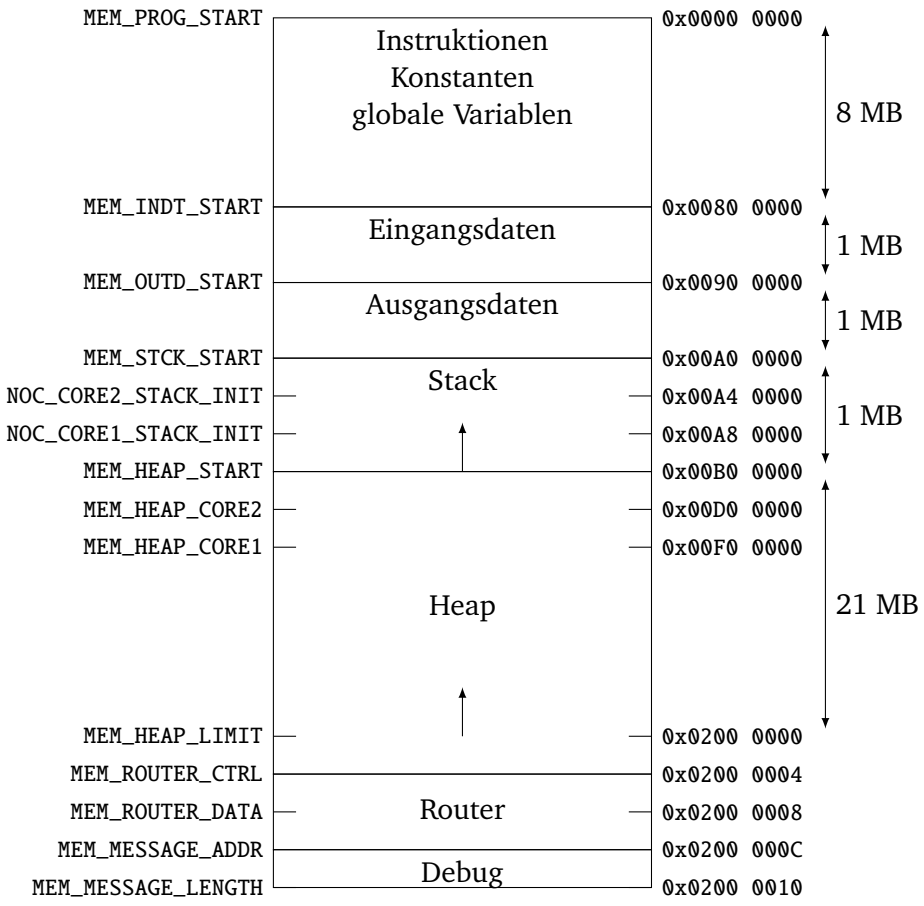


Abbildung 4.3: Aufteilung des Adressraums, die Pfeile zeigen die Größenzunahme der dynamischen Abschnitte.

Heap musste deswegen implementiert werden. Der Speicher kann über vordefinierte Adressen direkt angesprochen werden. Die Größen der jeweiligen Bereiche können der Abbildung 4.3 entnommen werden.

Die Größen der einzelnen Bereiche waren unterschiedlich motiviert. Die erste Adresse der Eingangsdaten und damit der Speicherbereich für die Instruktionen ist auf das Konzept des 3D-DRAM zurückzuführen. Die latenzoptimierte Schicht (Abschnitt 3.3.3) verfügte über 8 MB Kapazität. Da die Eingangsdaten erst nach dieser Schicht begannen, konnte diese Schicht für weitere Funktionalitäten verwendet werden (Abschnitt 5.4).

Die Größe der Eingangsdaten ist auf 1 MB beschränkt, da Simulationen von Daten dieses Umfangs etwa zwei Wochen realer Simulationszeit in Anspruch nahmen. Dieser Zeitabschnitt wurde als noch akzeptabel für den Rahmen dieser Arbeit eingestuft, sodass größere Datenmengen auf eine andere Art emuliert wurden (Abschnitt 5.2.3). Bei den Testapplikationen dieser Arbeit handelt es sich um Implementierungen von Algorithmen, die eine große Datenmenge in eine kleinere Datenmenge verwandeln. Daher waren 1 MB für die Ausgangsdaten in Bezug auf die Eingangsdaten ausreichend.

Während der Ausführung einer Applikation werden der Heap und der Stack intensiv für Datenstrukturen genutzt, sodass ihre Größen in Abhängigkeit von der Größe der Eingangsdaten zunehmen. Empirische Untersuchungen an den Applikationen zeigten, dass die vorgestellten Größen für diese Eingangsmenge ausreichend waren. Die limitierenden Faktoren waren hierbei die Möglichkeit des Simulators, solche Mengen zu verwalten, sowie die zur Verfügung stehenden Kapazitäten des Servers.

Der letzte Abschnitt des Speichers mit Debugging- und Routeradressen erweitern ihn soweit, dass während der Ausführung ein Zugriff in diesen Bereich auch ausgeführt werden kann. Der Simulator würde sonst einen Fehler ausgeben, wenn die Datenadresse außerhalb des Adressbereichs des Speichers

liegt. Die Verarbeitung der Zugriffe auf diesen Bereich fand innerhalb der Testumgebung und nicht mehr im Speicher statt.

Die implementierten Funktionen der Speicherverwaltung betrafen ausschließlich den *Heap*. Mittels der entsprechenden C-Funktion `malloc` konnte Speicherplatz für Datenstrukturen auf dem Heap reserviert werden und mit `free` wieder freigegeben werden. Für die Verwaltung wurde eine spezielle (Ausschnitt 4.2.1) Datenstruktur am Anfang des jeweiligen Abschnitts erzeugt.

4.2.1 Datenstruktur für Verwaltung des Heaps

```
1 typedef struct {
2   unsigned int  size;          // size of this memory tile
3   unsigned char valid;         // data validity
4 } t_memory_tile;
```

Der aktuell freie Speicherplatz wurde durch eine globale Variable *heap pointer* angezeigt. Bei einem Mehrkernsystem wurde der *Heap* noch weiter aufgeteilt, sodass jeder Kern über einen eigenen *heap pointer* verfügte. Dadurch konnten eventuelle Datenabhängigkeiten aufgelöst werden. Bei einer Reservierung eines Speicherabschnitts wurde der *heap pointer* um eine Werte, der sich aus der angefragten Größe und der Größe der Verwaltungsstruktur zusammensetzt, verringert und das `valid`-Byte auf `1` gesetzt. Die Freigabefunktion setzte diesen Wert auf `0` und überprüfte, ob der *heap pointer* wieder erhöht werden kann. Weitere Funktionen wie `calloc`, `realloc` und `memalign` griffen im Kern auf die `malloc` und die `free` zu.

Eine Auswertung der Veränderung des *heap pointers* während der Ausführung einer Applikation ergab, dass dieser Zeiger beim Abschluss der Verarbeitung nicht den Ursprungswert annahm. Da die Eingangsdaten erfolgreich verarbeitet wurden, ist davon auszugehen, dass bestimmte Datenstrukturen von der Applikation zwar reserviert, aber nie freigegeben wurden.

Folgende Liste zeigt alle implementierten Speicherverwaltungsfunktionen.

4.2.1 Implementierte Speicherverwaltungsfunktionen

```
void * malloc(                  size_t size);
void * calloc(  size_t num,     size_t size);
void * realloc( void * ptr,     size_t size);
void   free(    void * ptr);
void * memalign(size_t alignment, size_t size);
```

4.2.2 Datenein- und ausgabe

Die Abbildung 4.3 zeigt, dass die Ein- und Ausgangsdaten jeweils eigene Abschnitte im Adressraum darstellen. Dennoch ist ihre Verwaltung vom Konzept her anders aufgesetzt als die Verwaltung des *Heap*. Die Applikationen gehen davon aus, dass die Daten als Dateien vorliegen. Sie greifen auf sie mittels der entsprechenden Funktionen zu. Die Implementierung des Zeigers auf eine Datei ist in Ausschnitt 4.2.2 dargestellt.

4.2.2 Datenstruktur zur Dateienverwaltung

```
typedef struct{
  unsigned int    mode;           // access mode
  unsigned int    byte_index;     // next byte to access
  unsigned char * byte_pointer;   // next address to access
  unsigned char * file_start;     // first address
  unsigned int    file_length;    // current file length
} FILE;
```

Die mode-Variable kann nur zwei Werte (‚0' für Lesen und ‚1' für Schreiben) annehmen. Sie identifiziert somit jeweils Ein- oder Ausgabe. Mit dem `byte_index` wird die aktuelle Bytenummer gespeichert, und der `byte_pointer` zeigt auf die entsprechende Speicheradresse. Die (beim Schreiben veränderbare) Dateigröße wird in der Variable `file_length` erfasst, die zusammen mit `file_start` eine Setzung des `byte_pointers` durch die Applikation mithilfe der Funktion `fseek` erlaubt. Neben den stream-basierten Lese- und Schreibfunktionen sind unter anderem auch die byte-basierten Varianten implementiert.

Folgende Tabelle fasst alle implementierten Funktionen zusammen:

4.2.2 Implementierte I/O Funktionen

```
FILE * fopen( const char * filename, const char * mode);
size_t fread(       void * ptr, size_t size, size_t nmemb, FILE * stream);
size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream);

int      fseek( FILE * stream, long int offset, int origin);
long int ftell( FILE * stream);

int fputc( int character, FILE * stream);
int getc(                 FILE * stream);

int fscanf(  FILE * stream, const char * format, ...);
int fprintf( FILE * stream, const char * format, ...);
```

4.2.3 Netzwerkschnittstelle

Das Testsystem mit mehr als einem Kern verfügte über ein Routernetzwerk, dass zusammen mit dem gemeinsam genutzten Speicher für Datenübertragungen verwendet werden konnte. (Eine genaue Beschreibung ist im Abschnitt 3.2 zu finden). Um der Applikation einen Zugriff auf dieses Netzwerk zu ermöglichen, war eine entsprechende Schnittstelle erforderlich:

4.2.3 Netzwerkschnittstelle

```
int router_send( unsigned int addr, void * data);

int router_send_ack();
int router_get_ack();
```

Die Schnittstelle umfasst hauptsächlich die Sendefunktion. Diese Funktion erwartet die Zielkernnummer und einen Zeiger auf eine beliebige Datenstruktur. Dieser Zeiger wird über das Routernetzwerk übertragen. Die Funktion wertet zunächst das send-Bit des Statusregisters des Routers aus und wartet, solange es gesetzt ist. Danach werden die Zieladresse und die Daten in die entsprechenden Register

übertragen, und der Sendeprozess wird gestartet. Bestätigungsfunktionen senden beziehungsweise überprüfen die Übertragung einer `0`.

Sowohl die Sende- als auch die Empfangsroutinen warten im ersten Schritt in einer Endlosschleife, bis die entsprechenden Statusbits freigegeben beziehungsweise gesetzt wurden. Diese Leseanfragen erzeugen permanente Netzwerkzugriffe. Daher wurde im Rahmen dieser Arbeit die Möglichkeit getestet, die Informationen über einen Datenempfang mithilfe des Konzepts der Prozessorunterbrechungen zu implementieren. Der Parallelisierungsansatz dieser Arbeit setzt Synchronisationspunkte und einen Hauptstrang voraus. In einem Synchronisationspunkt konnte das dauerhafte Auslesen des Statusregisters eines Routers durch Unterbrechungsfunktionen nicht vermieden werden [Bied, 2014]. Daher wurde die Unterbrechungslogik in der finalen Version der Arbeit wieder entfernt.

4.2.4 Mathematische Funktionen

Für die Kernberechnungen verfügten die Testapplikationen über eigene Routinen. Dennoch konnte eine Reihe an mathematischen Funktionen als aufgerufen und relevant eingestuft werden. Auflistung 4.2.4 zeigt diese Menge:

4.2.4 Implementierte mathematische Funktionen

```
int    abs(   int    n );
double fabs(  double x );

double floor( double x );
double ceil(  double x );
double round( double arg);

double sqrt(  double x );
double pow(   double base, double exponent );
```

Berechnungen einer Quadratwurzel und einer Potenz sind dabei die aufwendigsten. Die Potenz wurde auf wiederholtes Multiplizieren zurückgeführt. Die Quadratwurzel wurde mithilfe des angepassten Newton-Verfahrens berechnet.

4.2.5 Konsolenausgaben

Eine weitere Betriebssystemfunktionalität stellen Konsolenausgaben dar. Diese Form der Nutzerinformation ist für die Ausführung einer Applikation auf den ersten Blick irrelevant. Sämtliche Ausgaben von Statusinformationen sind nur von Bedeutung, wenn die Ausführung fehlerhaft war. Der Portierungsprozess und die Tatsache, dass einige Funktionen neu implementiert werden mussten, verursachte Fehler. Darum wurden die Konsolenausgaben auch innerhalb der Testumgebung benötigt.

Bei den Statusausgaben lassen sich zwei Implementierungsstufen unterscheiden. In der einfachsten Form beinhaltet eine solche Nachricht lediglich einen Text, der über erfolgreich abgeschlossene Prozesse informiert oder deren Initialisierung andeutet. Wesentlich schwieriger zu implementieren sind Nachrichten, die neben einem Text auch Variablenwerte enthalten. Während die erste Form bei einer Fehlersuche nur Ansätze liefern kann, kann die zweite Form zu konkreten Anhaltspunkten führen. Daher wurden beide Varianten in der Testumgebung implementiert. Für Statusausgaben mithilfe einer Standardfunktion, wie sie von Applikationen genutzt wird, ist die reine Textausgabe vorgesehen. Die Implementierung baut auf der Funktion `printf` auf. Ihre Schnittstelle ist wie folgt definiert:

```
int printf( const char * format, ... );
```

Die Funktion bekommt also einen Zeiger auf den Text. Der Rest der Übergabeparameter wird ignoriert. Die Simulation bietet zwei mögliche Varianten einer Konsolenausgabe:

- `report`-Funktion in VHDL
- `echo`-Funktion in Tcl

Eine empirische Untersuchung zeigte, dass Auswertungen durch Tcl-Funktionen signifikant mehr realer Simulationszeit beanspruchen als vergleichbare Implementierungen in VHDL. Da der Textinhalt ohnehin innerhalb des simulierten Speichers zu finden war, bot eine Auswertung direkt in VHDL weitere Vorteile. Die Aufgabe bestand darin, der VHDL-Testumgebung während der Simulation mitzuteilen, dass die simulierte Software eine Konsolenausgabe initiiert.

Eine mögliche Schnittstelle ist in den speziellen Assemblerbefehlen zu finden, die bei einer regulären Programmausführung nicht benötigt werden: in Systemaufrufen. Diese Befehle können laut der MIPS-Spezifikation mit einem Parameter versehen werden, sodass innerhalb eines Systemaufrufs noch weiter differenziert werden kann. Die VHDL-Testumgebung wurde um eine globale Variable erweitert. Der Decoder des Prozessors setzte entsprechende Signale bei Systemaufrufen. Ausschnitt 4.2.5 zeigt die Implementierung:

4.2.5 Modellierung der Behandlung von Systemaufrufen

```
1 --synthesis translate_off
2   when MIPS_FUNC_SYSCALL  =>
3     case( instr_in(10 downto 6) ) is
4       when b"00000" => i_sim_control.sim_message    <= '1';     -- debug messages
5       when b"00001" => i_sim_control.sim_stop       <= '1';     -- simulation stop
6       when b"00010" => i_sim_control.print_message  <= '1';     -- printf messages
7       when b"00011" => i_sim_control.sim_finish     <= '1';     -- regular exit
8       when others   =>
9     end case;
10 --synthesis translate_on
```

Die Funktion `printf` führt einen Systemaufruf mit dem Parameter 2 (binär `00010`) aus. Durch diesen Aufruf wird das entsprechende Signal gesetzt (Zeile 6), und die VHDL-Testumgebung registriert eine Konsolenausgabe. Die weitere Aufgabe besteht darin, der Umgebung mitzuteilen, an welcher Speicheradresse der Text zu finden ist. Außerdem wird übermittelt, wie lang der Text ist. Dafür wurden spezielle, in Abbildung 4.3 bereits dargestellte Speicherstellen reserviert (`MEM_MESSAGE_ADDR` und `MEM_MESSAGE_LENGTH`). Die Funktion `printf` überträgt vor dem Systemaufruf den Zeiger und die Nachrichtenlänge an diese Adressen. Die VHDL-Testumgebung leitet die Anfrage an den Speicher weiter, der die Nachricht wiederum Zeichen für Zeichen zusammensetzt und über eine `report`-Funktion ausgibt. Diese Erweiterung wurde nur im Modell des idealen Speichers implementiert, da während der Portierung nur dieser Speicher genutzt wurde.

Für eine Ausgabe von Variablenwerten wird zwar der gleiche Kommunikationsweg zwischen der Applikation und der VHDL-Testumgebung benutzt, darüber hinaus unterscheidet sich diese Methode jedoch. Im ersten Schritt muss eine Variable gekennzeichnet werden. Das erfolgt mithilfe des folgenden Konstrukts:

```
//sim_message("Info Text ", variable)
```

Der Ausdruck wird bei einem regulären Übersetzungsprozess als ein Kommentar gewertet und ignoriert. Dieses Verhalten ist gewünscht, um die Ausgaben nach der Fehlersuche zu unterdrücken. Während

der Fehlersuche wird die Quellcodedatei durch ein spezielles Script nach diesem Konstrukt durchsucht und in folgende Zeile verwandelt:

```
sim_message(2, variable); //sim_message("Info Text ", variable)
```

Ähnlich wie bei der ersten Methode wird eine Funktion (hier `sim_message`) aufgerufen. Dabei bekommt sie zwei Übergabeparameter. Der erste Parameter enthält eine Nummer, unter der der Ausgabentext zu finden ist. Der Text wird separat behandelt. Nachrichten von allen solchen Ausgaben werden in einer Tabelle zusammengefasst. Der zweite Übergabeparameter ist der Variablenwert. Die Funktion `sim_message` signalisiert - wie schon bei `printf` - über einen Systemaufruf einen entsprechenden Vorgang (Zeile 4). Das Signal wird in diesem Fall nicht durch die VHDL-Testumgebung erfasst, sondern durch einen Tcl-Script als Simulationsbegleitung. Diese Tcl-Scripte werden im Abschnitt 4.3.2 beschrieben. Das Script greift direkt auf die Registerbank des Prozessors zu und entnimmt die Werte der Übergabeparameter. Die Nummer des Textes führt in der Nachrichtentabelle zum entsprechenden Eintrag. Der Variablenwert wird direkt aus dem reservierten Register (in MIPS `a1`) ausgelesen.

4.2.6 Bootvorgang

Nach einem Reset-Signal befindet sich ein System in einem definierten Initialzustand. Der Bootvorgang sorgt dafür, dass die Ausführung einer Applikation aus diesem Zustand heraus beginnen kann. Für diese Arbeit wurde eine möglichst kurze Form des Bootsvorgangs implementiert, wie folgender Ausschnitt 4.2.6 zeigt:

4.2.6 Assemblerkode des Bootvorgangs auf einem Mehrkernsystem mit zwei Kernen

```
1   la   $gp, _gp                    // 1. set global pointer
2   lw   $a0, 0x2000004              // load core number from router
3   srl  $a0, $a0, 28                // extract core number
4
5  init_core0: la $a1, 0x0           // check for core number 0
6   bne $a0, $a1, init_core1         // if not got to core number 1
7   la   $sp, 0xB00000               // 2. set stack pointer
8   j    main_enc                    // 3. go to main function
9
10 init_core1: la $a1, 0x1           // check for core number 1
11  bne $a0, $a1, init_core2         // if not go to core number 2
12  la   $sp, 0xA80000               // set stack pointer
13  j    main_noc                    // go to main function
14
15 init_core2: break 0x0             // undefined core number
```

Bei einem Mehrkernsystem bekommt jeder Kern eine eigene Hauptfunktion und den Initialwert des *stack pointers*. Das Statusregister des Routers enthält die jeweilige Kernnummer, sodass anhand dieser Nummer die Zuweisung erfolgen kann. Auf einem System mit nur einem Kern entfallen diese Verzweigungen, sodass der Bootvorgang auf drei Befehle reduziert werden kann (die Zeilen 1, 7 und 8).

4.3 Unterstützende Softwareumgebung

Bei der Beschreibung der Betriebssystemfunktionalität wurde der Einsatz von zusätzlichen Scripten bereits angedeutet. Ihre Aufgabe bestand darin, den Testprozess zu automatisieren. Eine Reihe dieser Scripte unterstützte die Erzeugung von Testfällen. Eine weitere Reihe unterstützte den Simulationsprozess. Der überwiegende Teil der Scripte ist in der Sprache Perl implementiert, der Rest in Tcl.

4.3.1 Testfallgenerierung

Eine Aufgabe, die viel Automatisierungspotenzial bietet, ist die Testfallerzeugung. Dabei geht es darum, den Quellkode einer Applikation in eine Form zu verwandeln, die auf dem simulierten Prozessor ausgeführt werden kann. Außerdem müssen die Eingangsdaten entsprechend aufbereitet werden. Eine getrennte Erzeugung dieser beiden Bestandteile ist auch vorteilhaft. Neben der eigentlichen Applikation werden weitere Daten für eine eventuelle Fehlersuche und für die Speichernutzungsanalyse gebraucht. Der Erzeugungsprozess sollte also Ansätze dafür bieten, diese Daten generieren zu können. Nicht zuletzt sollten die Eingangsdaten für alle getesteten Speichermodelle verwendet werden können. Außerdem muss die Ausgabe nach der Simulation ausgewertet werden können.

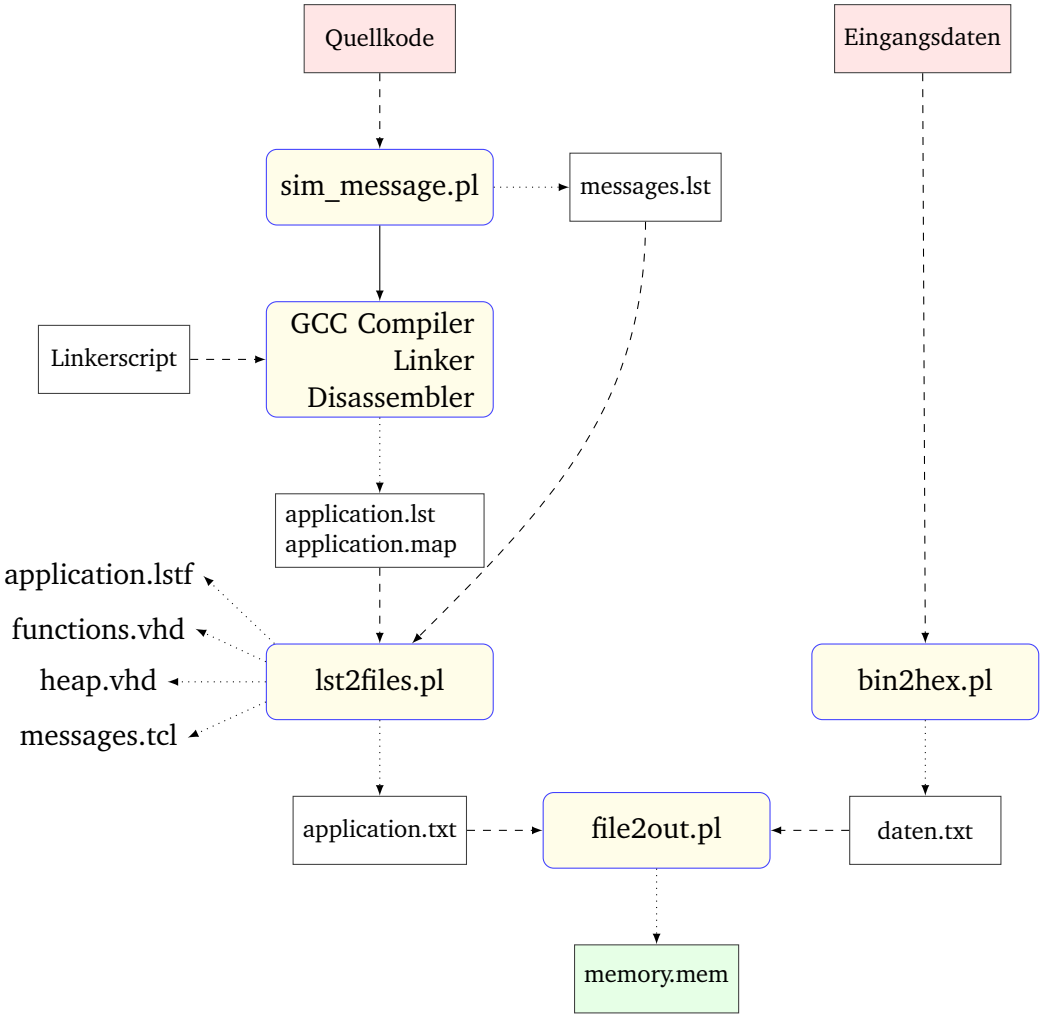


Abbildung 4.4: Ablaufdiagramm der Testfallerzeugung

Alle diese Kriterien führten zu einem Ablauf, wie er in der Abbildung 4.4 dargestellt ist. Der Quellkode einer Applikation wird im ersten optionalen Schritt nach Konsolenausgaben durchsucht (beschrieben im Abschnitt 4.2.5). Aus dem Ergebnis wird eine vorläufige Tabelle (`message.lst`) generiert, die alle gefundenen Texte enthält. Danach folgt der Übersetzungsschritt durch einen GCC-Compiler. Anschließend werden alle so erzeugten Objektdateien mit Betriebssystemfunktionen verlinkt. Dieser Schritt kann auch durch einen Linkerscript gesteuert ablaufen.

Die Ausführung einer Applikation auf einem simulierten Prozessor lässt auf der einen Seite tief bis auf Registerwerte und Pipelinefluss blicken, auf der anderen Seite stehen keine Debugginginstrumente zur Verfügung, um die Abarbeitung vom C-Kode nachverfolgen zu können. Dazu ist eine Schnitt-

stelle erforderlich, die Anfragen von einem möglichen Debugger an die ModelSIM-Umgebung weiterleitet. Dieser Hardware/Software-Kosimulationsansatz ist Gegenstand aktueller Forschungsarbeiten [Willenberg und Chow, 2013]. Es existierte zum Zeitpunkt dieser Arbeit keine allgemeine und frei verfügbare Lösung, sodass diese Aufgabe durch vorhandene Mittel angegangen werden musste. Die größte Information enthielt eine disassemblierte Version des Instruktionscodes. Diese Aufschlüsselung der Applikation stellte zwar keine echte Alternative für einen Debugger dar, da die Zuordnung von C-Variablen auf das CPU-Register fehlte. Dennoch konnten damit viele Fehler gefunden werden.

Der nächste Schritt umfasst die Erzeugung aller notwendigen Dateien, sodass nur noch die Eingangsdaten angehängt werden müssen. Das Script *lst2files.pl* erzeugt

- *application.lstf*: Assemblerkode mit Einsprungmarken für C-Funktionen,
- *functions.vhd*: Ausgabe der Funktionennamen während der Ausführung,
- *heap.vhd*: Ausgabe der Wertänderung des *heap pointer*,
- *messages.tcl*: Tcl-Script für die Konsolenausgabe der Variablenwerte,
- *application.txt*: Hexcode der Ausführungsdatei der Applikation, aufgefüllt mit Nullen bis zur Adresse `0x0080 0000`.

Die Eingangsdaten werden lediglich von einer binären Form in eine Textdatei umgewandelt und an den Applikationskode angehängt. Das Script *file2out.pl* übernimmt die Umwandlung in eine Form, die für jedes getestete Speichermodell verwendet werden kann. Als Ausgabe wird bei einem idealen Speichermodell eine speziell formatierte Textdatei erzeugt. Für ein 3D-DRAM-Modell wird eine Datei pro Schicht generiert. Das Laden dieser Dateien während der Simulation übernehmen die Tcl-Scripte (Abschnitt 4.3.3). Die getrennte Erzeugung von Applikations- und Eingangsdaten erlaubt eine Variation der Eingangsdaten und damit variable Untersuchungen.

Nach der Ausführung der Applikation wird die Simulation nicht sofort beendet. Der Ausgabeabschnitt des Speichers wird zuvor in eine Ausgabedatei geschrieben. Diese Datei kann mithilfe des *file2out.pl*-Scripts in eine Textform für genaue Betrachtung umgewandelt werden oder in eine binäre Form für direkte Ansicht des entsprechenden Dateiformats auf dem Arbeitsrechner.

4.3.2 Scripte für Hardwareentwurf

Weitere Scripte unterstützten die Simulation der erzeugten Testfälle. Ihr Einsatzbereich erstreckte sich allerdings über diese Aufgabe hinaus. Die komplette Verwaltung, Anpassung, Simulation, Verifikation und Synthese von digitalen Hardwarekomponenten konnte mithilfe dieser Scripte bewältigt werden. Dabei kann der entsprechende Prozess über eine Konfigurationsdatei gesteuert werden. Diese Konfigurationsdatei enthält Signalwörter und -zeichen, die von den Scripten ausgewertet werden. Die Struktur dieser Scripte ist in Abbildung 4.5 dargestellt.

Im Gegensatz zur Testfallgenerierung werden alle Perl-Simulationsscripte über das Hauptprogramm *crate_design.pl* gesteuert. Dieses Programm kann über Nutzerangaben gesteuert werden und verwaltet die registrierten Entwürfe. Dabei liest das Programm den Hauptabschnitt der Konfigurationsdatei (gekennzeichnet mit `//ENV`) und führt weitere Scripte aus. Diese Scripte lesen die für sie wichtigen Abschnitte der Konfigurationsdatei und erzeugen Ausgaben für eine weitere Verarbeitung. Im Folgenden werden diese Abschnitte genauer erläutert.

Abschnitt `ENV`

Der erste Abschnitt der Konfigurationsdatei enthält Informationen über die Aufgabe, die die Scripte unterstützen sollen. Eine Beispielkonfiguration ist im Ausschnitt 4.3.2 dargestellt.

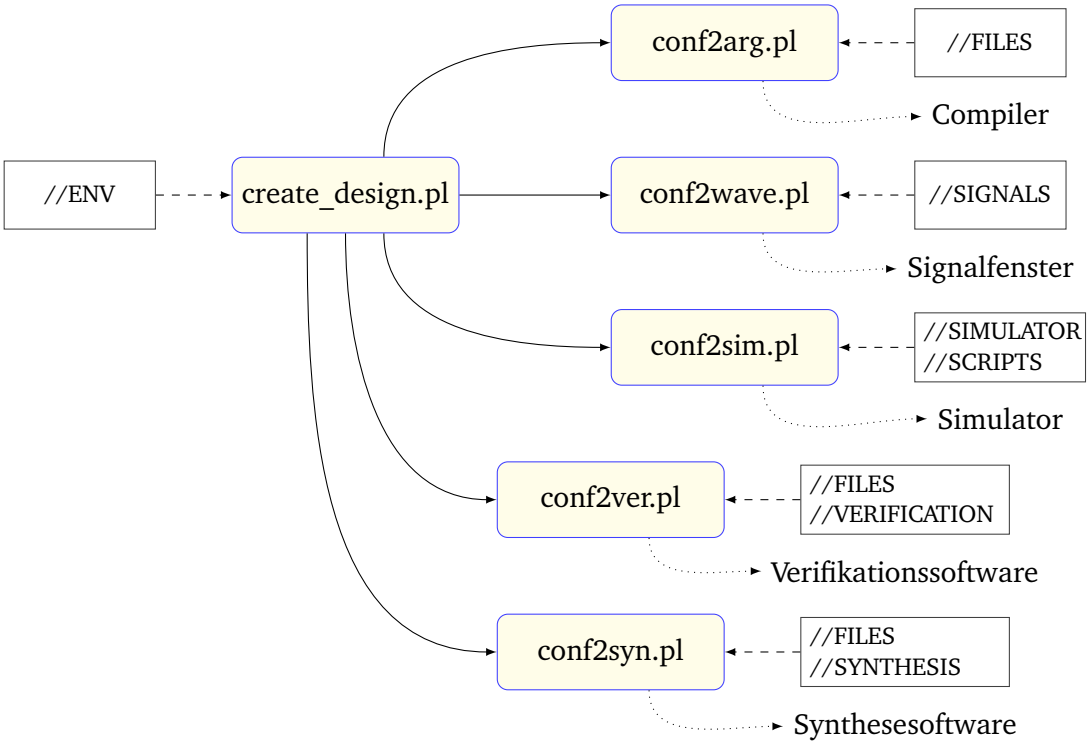


Abbildung 4.5: Struktur der Simulationsscripte. Die mit // gekennzeichneten Wörter entsprechen den Abschnitten in einer .conf-Datei.

4.3.2 Beispiel für den Inhalt des Abschnitts `ENV`

```
1 //ENV
2 -- #WORK      /hardware/lib  -- library creation
3 #SIM         ModelSim       -- simulation
4 #VER         OneSpin        -- formal verification
5 #SYN         Synplify       -- netlist synthesis
```

Die Kommentare werden wie bei VHDL mit - - eingeleitet. Deshalb wird die Zeile 2 als auskommentiert gewertet. Alle Signalwörter innerhalb eines Bereichs werden mit # gekennzeichnet. Die Zeilen 3 bis 5 benennen einen Arbeitsordner, in dem die entsprechenden Arbeitsvorgänge eingeleitet werden. Sie geben Aufschluss darüber, mit welchen CAD-Werkzeugen diese Arbeitsvorgängen bewerkstelligt werden. Falls ein Ordner nicht existiert, wird er vom *create_design.pl* angelegt.

Das Signalwort `WORK` gibt an, dass eine Arbeitsbibliothek erzeugt werden soll. Diese Information kollidiert mit der Simulationsanweisung (mittels `SIM`). Daher ist sie auskommentiert. Eine Simulation erwartet eine Datei mit einem Testmodul, während eine Bibliothek nur ein Hauptmodul benötigt.

Abschnitt `FILES`

In diesem Abschnitt werden alle benötigten Dateien und Information für die jeweiligen Compiler verwaltet. Ausschnitt 4.3.2 zeigt eine Beispielkonfiguration:

4.3.2 Beispiel für den Inhalt des Abschnitts FILES

```
1 //FILES
2 #L work                                                    -- working library
3 #C vcom                                                    -- compiler command
4 #P -explicit -93                                           -- compiler parameter
5
6 #VAR src_path         ../../cpu                            -- variable
7
8 ${src_path}/decoder.vhd #P -O5                             -- source file
9 #T ${src_path}/cpu.vhd                                     -- top module
10 #TB ${src_path}/testbenches/tb_cpu.vhd                    -- testbench
11
12 #INCL env.link                                            -- additional file
13 #LINK ${src_path}/../memory/memory_in.mem  memory_in.mem  -- link file
```

In diesem Abschnitt werden als Erstes die Arbeitsbibliothek, der Compiler und seine Eingabeparameter definiert. Anschließend erwartet das Script Dateien, die das Modell beschreiben. Dabei können Variablen für Ordner oder sonstige Bezeichnungen deklariert werden. Dabei ist der Arbeitsordner des Scripts der Ort der Konfigurationsdatei. Die Dateien werden in der Reihenfolge compiliert, in der sie im Abschnitt notiert wurden. Einzelnen Dateien können auch separate Compileroptionen oder Bibliotheksnamen zugeordnet werden. Darüber hinaus können sie als Hauptmodul oder Testdatei gekennzeichnet werden.

Des Weiteren erlaubt das Script *conf2arg.pl* das Einfügen weiterer Dateien, deren Inhalt nach den gleichen Regeln ausgewertet wird. So können sich wiederholende Konfigurationen ausgelagert werden. Die eingefügten Dateien können allerdings selbst keine weiteren Dateien aufnehmen. Darüber hinaus besteht die Möglichkeit, symbolische Verknüpfungen mithilfe des Scripts anzulegen, wobei diese Angaben im Prozessablauf als Erstes ausgeführt werden. So können weitere Scripte direkt mit diesen Verknüpfungen arbeiten.

Abschnitt SIMULATOR

Die Angaben für die Simulation enthält der Abschnitt SIMULATOR. Eine Beispielkonfiguration ist im Folgenden Ausschnitt zu sehen:

4.3.2 Beispiel für den Inhalt des Abschnitts SIMULATOR

```
1 //SIMULATOR
2 #S vsim                                                    -- simulator
3 #P -t ps -vopt                                             -- options
4
5 #INCL env.vopt                                             -- additional file
6
7 #T tb_cpu                                                  -- testmodul
8 #R -all                                                    -- runtime
9 -- #R 100 ns
```

Das Script *conf2sim.pl* benötigt die Angaben zum verwendeten Simulator (hier ModelSim-Befehl) und seine Übergabeparameter. Diese Parameter können ebenfalls in einer externen Datei enthalten sein. Des Weiteren werden das Testmodul und die Simulationszeit benötigt. Diese Zeit kann auch durch die Testumgebung bestimmt werden (wie im Beispiel durch -all).

Abschnitt SCRIPTS

Während der Simulation werden noch weitere Tcl-Scripte geladen. Sie werden im Abschnitt 4.3.3 näher beschrieben. Ihre Einbindung wird durch den Abschnitt SCRIPTS gesteuert:

4.3.2 Beispiel für den Inhalt des Abschnitts SCRIPTS

```
1 //SCRIPTS
2 #CONST MEMORY   0                                    -- Tcl constant
3 #CONST constants.tcl                                 -- script with constants
4
5 memory.tcl                                           -- source script
```

In diesem Abschnitt wird nur zwischen Quellenscripten und Tcl-Konstanten unterschieden. Die letzteren werden durch das Tcl-Befehl `set` vor dem Start des Simulators geladen, während die Quellenscripte nach dem Start des Simulators mit dem `source`-Befehl eingebunden werden.

Abschnitt SIGNALS

Während des Hardwareentwurfsprozesses und insbesondere für eine Fehlersuche ermöglicht ModelSIM eine Verfolgung der Signalwertänderung in einem entsprechenden Fenster. Die Signale können dabei umbenannt, in Gruppen zusammengefasst und in unterschiedlichen Formaten dargestellt werden. Das Script *conf2wave.pl* wertet dazu den Abschnitt SIGNALS aus und erstellt eine Konfigurationsdatei für ModelSIM. Diese Datei enthält die nötigen Darstellungsinformationen. Darüber hinaus kann das Signalfenster in weitere Unterabschnitte aufgeteilt werden, sodass die Signale von Untermodulen eines Entwurfs intuitiv einsortiert werden können. Der Ausschnitt 4.3.2 zeigt ein Beispiel:

4.3.2 Beispiel für den Inhalt des Abschnitts SIGNALS

```
1 //SIGNALS
2 #U /tb_cpu                              -- current path
3 #N 1                                    -- show names only
4 #Z 0 100                                -- zoom window
5 #R ns                                   -- resolution
6
7 #G GENERAL                              -- begin of group with name
8  clk                                    -- signal, default format
9 #BIN wr_mask                            -- signal, binary format
10 #HEX instr_addr                        -- signal, hex format
11 uut/cpu/datapath/regbank #N regbank    -- signal, renamed
12 #                                      -- end of group
13
14 #U /tb_cpu/uut
15 #INCL cpu.wave                         -- additional file
16
17 #P                                     -- new panel
18 #D NEW_PANEL                           -- label
```

Zunächst erwartet das Script eine Angabe über den aktuellen Pfad, die jederzeit geändert werden kann. Außerdem kann eine Unterdrückung der Pfadangabe in der Darstellung im Signalfenster aktiviert werden. Darüber hinaus können die relevante Simulationszeit und die Auflösung angegeben werden. Die Signale der Untermodule können in einer separaten Datei zusammengefasst und bei Bedarf geladen werden.

Abschnitt VERIFICATION

Neben der Simulation eines Hardwaremoduls besteht die Möglichkeit, seine Umsetzung mathematisch mit den geforderten Funktionen abzugleichen. Diese formale Verifikation ist in der Lage, alle möglichen Zustände zu überprüfen, ohne sie einzeln durchgehen zu müssen. Auf diese Weise ist es möglich, dem exponentiellen Wachstum der Zustandsmenge zu entkommen.

Im Rahmen dieser Arbeit wurde der Prozessor formal verifiziert [Chehab, 2015]. Das Script *conf2ver.pl* nimmt die Dateienliste aus dem Bereich `FILES` und steuert die CAD-Software für formale Verifikation über Angaben im Bereich `VERIFICATION` der Konfigurationsdatei. Ein Beispiel für Angaben in diesem Bereich gibt der folgende Ausschnitt:

4.3.2 Beispiel für den Inhalt des Abschnitts `VERIFICATION`

```
1 //VERIFICATION
2 #EO -vhdl_generic {core_idx=0 DEBUG_FLAG="OF"}  -- compilation parameter
3 #EP -top cpu                                    -- elaboration parameter
4
5 cpu.vhi                                         -- property file
```

Für diese Arbeit wurde die Software OneSpin [OneSpin Solutions, 2014] verwendet. Die Angaben während der Konfigurationsphase des Verifikationsprozesses sind als `compilation`- und `elaboration`-Parameter gekennzeichnet. Der konkrete Ablauf dieses Prozesses hängt starkt davon abhängig ab, welche Software benutzt wird. Die Verarbeitung durch das Script fällt entsprechend speziell aus. Die Einbindung der Eigenschaftsdatei ist zwar allgemein gehalten, deren Inhalt ist jedoch stark an Vorgaben von OneSpin gebunden.

Abschnitt SYNTHESIS

Der Übergang von der Simulation hin zu einem Entwurf, der auf einem FPGA ausgeführt werden kann, wird ebenfalls teilweise durch die Scripte unterstützt. Insbesondere die Netzlistenerzeugung ist die Hauptaufgabe des *conf2syn.pl*-Scripts:

4.3.2 Beispiel für den Inhalt des Abschnitts `SYNTHESIS`

```
1 //SYNTHESIS
2 #C synplify_pro              -- synthesis tool
3
4 #P -technology  Virtex5      -- technology
5 #P -part        XC5VFX70T
6 #P -package     FF1136
7
8 #D netlist                   -- output folder
9 #O cpu.edf                   -- output file
10 #L cpu.srr                  -- report file
11 #
```

Neben den Angaben zur Zieltechnologie erwartet das Script den Namen des Arbeitsordners sowie den der Ausgangs- und der Protokolldatei.

4.3.3 Tcl Scripte

Der Simulator ModelSIM erlaubt es, Tcl-Scripte zu laden und während der Simulation mittels dieser Scripte auf bestimmte Ereignisse zu reagieren. In dieser Arbeit wurden drei Aufgaben mithilfe dieser Scripte erledigt:

- Laden und Zurückschreiben des Speicherinhalts
- Ausgabe der Variablenwerte.
- Umsetzung von Simulationsstopps über die Applikation und regulärer Abschluss der Simulation.

Spezielle ModelSIM-Befehle sind dafür konzipiert, den Inhalt eines simulierten Speichers zu laden und auszulesen. Zu Beginn der Simulation wird der Inhalt des Instruktions- und des Eingangsdatenabschnitts (Abbildung 4.3) direkt aus den entsprechenden Dateien geladen. Der simulierte Speicher enthält also alle Informationen, dass die Applikation mit der Bearbeitung beginnen kann. Am Ende der Bearbeitung wird der komplette Inhalt des Ausgabenabschnitts in dazugehörige Ausgabendateien geschrieben, sodass das Ergebnis der Ausführung begutachtet werden kann. Für das 3D-DRAM-Modell muss je eine Datei pro Schicht verwendet werden. Ausschnitt 4.3.3 zeigt die Implementierung der Ladefunktion für dieses Modell:

4.3.3 Laden des Inhalts des Speichers für das 3D-DRAM Model

```
for {set bank 0} {$bank < 8} {incr bank} {
  set name [list $INPUT_FILE $bank ".mem"]
    mem load -infile [join $name ""] -format hex -filltype value
                                                 -filldata 128'h0
                        $MEMORY_PATH/bank($bank)/bank_unit/memory
  }
```

Der Ansatz für die Ausgabe der Variablenwerte wurde im Abschnitt 4.2.5 bereits beschrieben. An dieser Stelle sei nur auf die konkrete Umsetzung in Tcl anhand eines Ausschnitts verwiesen.

4.3.3 Ausgabe eines Variablenwertes mittels einer Tcl-Funktion, Einkernsystem

```
when { sim_message = 1 } {
  set INDEX [examine -hex $REG_PREFIX/$REG_POSTFIX/mem_reg_bank(4)]
  set VALUE [examine -hex $REG_PREFIX/$REG_POSTFIX/mem_reg_bank(5)]

  print_message 0 $INDEX $VALUE
}
  ...
proc print_message { {core 0} {index "0"} {value "0"} } {
 global now
  switch -- $index {
      "32'h00000000" { echo $now ns: core $core : input size: $value }
      "32'h00000001" { echo $now ns: core $core : execution finished $value }
  }
}
```

Auf die gleiche Weise wie für Konsolenausgaben informierte die Applikation über eine spezielle Funktion mittels Systemaufrufen darüber, dass ein Simulationsstopp oder Simulationsende an einer bestimmten Stelle erwünscht ist. Die Änderungen der entsprechenden Variablen der VHDL-Testumgebung führten zu Tcl-Funktionen, die die Variablenwerte in Befehle `stop` oder `exit` übersetzten.

4.4 Sofwareportierung und -parallelisierung: Theoretischer Hintergrund

4.4.1 Portierung

Bis auf eine Implementierung in Assembler, die im Großen und Ganzen eine Menschen lesbare Abstraktion des Maschinenkodes darstellt, bieten alle höheren Programmiersprachen Konzepte an, die es erlauben, die gestellten Aufgaben plattformunabhängig zu lösen. Aus dieser Perspektive betrachtet, reduziert sich der Portierungsaufwand auf die Übersetzung durch den jeweiligen Compiler. Leider stellt sich in der Praxis heraus, dass viele Applikationen - schon durch die Aufgabenstellung - nicht auf die Kenntnis der ausführenden Hardware verzichten können. Die meisten Teilfunktionen bleiben zwar portabel, in ihrer Gesamtheit sind sie es jedoch nicht.

Dies führt dazu, dass die absolute Portabilität nie erreicht werden kann [Ford, 1977]. Es müssen Teile der Applikation geändert werden, sodass genau genommen nicht mehr von der gleichen Software gesprochen werden kann. Die absolute Softwaregleichheit als ein maßgebliches Kriterium erscheint allerdings zu restriktiv, wenn man bedenkt, dass sich die komplette Hardware geändert haben kann. Vielmehr stellt sich die Frage, inwiefern die geänderten Anteile einen Einfluss auf die Kernfunktion der Implementierung ausüben. Diese Kernfunktion kann - für sich betrachtet - ein durchaus hohes Grad an Portabilität aufweisen [Schneider, 1980].

Dieser Grad der Portabilität bestimmt letzten Endes der Programmierer selbst. In Abhängigkeit davon, wie die Variablen und sonstige Datencontainer deklariert werden, wie darauf zugegriffen wird und welche Aufgaben dem Compiler überlassen werden, kann jede Portabilitätsstufe erreicht werden. Die während der Portierung auftauchende Probleme lassen sich in der Regel auf genau diese drei Fragen zurückführen, wie die folgende Auflistung ohne Anspruch auf Vollständigkeit zusammenfasst [Hook und Gherman, 2006].

- **Register der Prozessoren**: Eine Variable vom Typ *integer* bedeutet in der Regel die volle Breite eines Prozessorregisters. Diese Breite unterscheidet sich aber vo Prozessor zu Prozessor und kann auch eine unterschiedliche Bytereihenfolge im Speicher bedeuten.

- **Fließkommazahlen**: Insbesondere in den Randbereichen der Darstellung von Fließkommazahlen können Implementierungsunterschiede auftauchen, die sich auf den Programmfluss auswirken können.

- **Skalierbarkeit**: Eine Software, die für leistungsstarke Systeme, die über hohe Rechen- und Speicherressourcen verfügen, implementiert wurde, kann Mikrocontroller oder sonstige auf Energieverbrauch optimierte Systeme überfordern.

- **Betriebssystem**: Wenn eine Applikation auf absolute Speicheradressen zugreift, können sie auf dem neuen System für andere Zwecke reserviert sein, sodass der Speicherzugriff im besten Fall ignoriert wird.

- **Compiler**: Eine Spezifikation ist nicht frei von Definitionslücken. Dies trifft auch auf die Programmiersprachen zu. Daher kann ein anderer Compiler bestimmte Konstrukte anders im Speicher ablegen. Wenn der Programmierer aber von einer bestimmten Anordnung ausgeht, kann dies auf dem neuen System zu schwerwiegenden und nur schwer aufzuspürenden Fehlern führen. Außerdem ist der Compiler auch „nur“ Software, die typische Softwarefehler aufweisen kann.

Alle diese Punkte können einen zusätzlichen Aufwand erforderlich machen, der im schlimmsten Fall zu einer kompletten Neuimplementierung führen kann. Insgesamt lässt sich schlussfolgern, dass die Portierung eine applikations- und hardwarespezifische Aufgabe bleibt.

4.4.2 Parallelisierung

Die Parallelisierung der Ausführung einer Applikation bekam eine starke Bedeutung, als die Zunahme der Frequenz durch die Zunahme der Anzahl der Prozessorkerne abgelöst wurde. Die Leistungsfähigkeit eines Systems hängt seitdem maßgeblich davon ab, wie ausgelastet die einzelnen Kerne sind. Dieser Umbruch fand ungefähr in der Mitte des ersten Jahrzehnts des 21. Jahrhunderts statt. Die Forschung über die Parallelisierung der Software hat allerdings schon viel früher angesetzt. Padura stellte bereits 1980 eine Klassifizierung der Datenabhängigkeiten und potenziell parallelisierbaren Schleifen vor. In diesem Zusammenhang präsentierte er Ansätze zu einer entsprechenden Compilierung [Padua et al., 1980].

Seitdem wurden viele Anstrengungen unternommen, die Parallelisierung möglichst automatisch zu realisieren. Das nach wie vor nur teilweise gelöste Problem stellen die Datenabhängigkeiten beziehungsweise deren Erkennung dar. Wenn die Datenabhängigkeiten im Vorfeld bekannt sind, lassen sie sich mit einer Reihe von Ansätzen auflösen. Ihre Erkennung erfordert allerdings eine Analyse, die erst dann die besten Ergebnisse liefert, wenn die Applikation währenddessen auch ausgeführt wird. Eine passive Analyse durch den Compiler ist zwar auch dazu in der Lage, diese Abhängigkeiten zu erkennen, die Ergebnisse sind allerdings zu pessimistisch, sodass das Potenzial der Leistungssteigerung nicht voll ausgenutzt wird [Karkowski und Corporaal, 1997]. Mithilfe des Gesetzes von Amdahl [Amdahl, 1967] lässt sich die potenzielle Beschleunigung errechnen. [Bridges et al., 2007] zeigte an einer großen Benchmarkreihe, dass die theoretischen Vorhersagen durchaus in der Praxis getroffen werden können.

Die Parallelisierung von Software ist nach wie vor ein großes Thema der Forschung und der aktuellen Publikationen. Insgesamt lässt sich festhalten, dass die Problemstellungen in diesem Bereich ähnlich wie bei der Portierung immer noch am besten durch einen Menschen gelöst werden kann und applikationsspezifische Ansätze erfordern. Bei dieser Arbeit stellte die parallele Software einen Testfall für die Speicher dar. Daher spielte die Parallelisierung nur eine untergeordnete Rolle, wie die speziellen Umsetzungen in den folgenden Kapiteln zeigen. Der verwendete Ansatz ist den Konzepten von [The OpenMP, 2014] entnommen und in Abbildung 4.6 beispielhaft für ein System mit drei Kernen veranschaulicht.

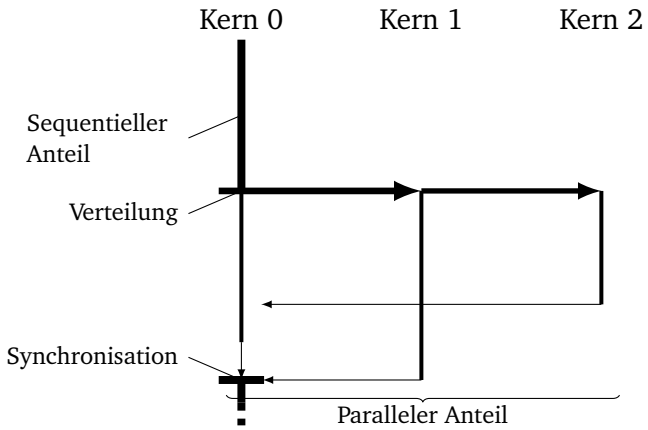


Abbildung 4.6: Ablauf der parallelen Ausführung einer Applikation auf einem System mit drei Kernen

Ein Kern übernimmt bei diesem Konzept den sequentiellen Teil und verteilt parallele Routinen an weitere Nebenkerne. Diese Kerne warten nur auf Anweisungen und signalisieren dem Hauptkern, sobald ihre Berechnungen abgeschlossen sind. Am Ende des parallelen Anteils existieren somit Synchronisationspunkte. Der Hauptkern wartet, bis alle anderen Kerne sich gemeldet haben. Der Ablauf kehrt zu einer sequentiellen Ausführung zurück und eine mögliche Verteilung kann aufs Neue beginnen.

4.5 Reale Applikationen

Bei den verwendeten Testapplikationen lassen sich zwei Kategorien unterscheiden. Zwei Funktionen sollen Grenzfälle der Speichernutzung repräsentieren und die restlichen Implementierungen stellen übliche Desktopanwendungen dar.

4.5.1 JPEG2000 - Bildkompression

Algorithmus

JPEG2000 ist eine Weiterentwicklung des JPEG-Standards und definiert Algorithmen zur Bildkompression. Der Standard bietet eine Reihe an Zusatzoptionen, beispielsweise Streamübertragung oder unterschiedliche Komprimierungsstufen für ausgewählte Bereiche. Die Komprimierungsartefakte führen dabei nicht wie bei JPEG zu „Kacheln“, sondern zu verschwommenen Bereichen, sodass der Inhalt der Bilder bei gleicher Dateigröße von einem menschlichen Auge als „besser“ eingestuft wird [JPEG, 2013].

Implementierung

Der Standard definiert eine Reihe von Funktionen, die bei einer konkreten Implementierung nicht zwangsläufig umgesetzt werden müssen. Daher existieren auch einige Implementierungen, die sich sowohl im Umfang als auch in der Lizenzierung unterscheiden.

Für diese Arbeit wurde eine volle, quelloffene Implementierung „openJPEG“ [OpenJPEG, 2014] verwendet. Diese Arbeit geht zurück auf eine Masterarbeit von David Janssens und wurde über die Zeit weiterentwickelt. Die Software ist modular aufgebaut, sodass die verschiedenen Konzepte des Standards optional eingebunden werden können. Für diese Arbeit wurde die in der Bibliothek `openjp2` implementierte Kernfunktion verwendet.

Portierung

Die Software ist vollständig in der Programmiersprache C geschrieben. Es gibt keine Routinen, die assemblerunterstützte Beschleunigungsansätze beinhalten. Die Portierung reduzierte sich auf eine Übersetzung durch den entsprechend konfigurierten Compiler. Allerdings lieferte die Ausführung der Applikation fehlerhafte Daten. Nach einer intensiven und tiefgreifenden Suche konnte ein Fehler in der Schreibfunktion `opj_write_bytes_BE` lokalisiert werden. Der folgende Ausschnitt zeigt die ursprüngliche und die geänderte Variante.

4.5.1 Änderung der `opj_write_bytes_BE` Funktion

```
1 // original version
2 const OPJ_BYTE * l_data_ptr = ((const OPJ_BYTE *) &p_value) + p_nb_bytes;
3 memcpy(p_buffer,l_data_ptr,p_nb_bytes);
4
5 // changed version
6 OPJ_UINT32 i;
7 for( i = 0; i < p_nb_bytes; i++ ){
8   p_buffer[i] = p_value >> 8*(p_nb_bytes - 1 - i);
9 }
```

Die Funktion `memcpy` ist Teil der neu implementierten Betriebssystemfunktionen und könnte ebenfalls fehlerhaft sein. Ihre Implementierung wurde sowohl einzeln getestet, als auch durch mehrere Aufrufe

innerhalb des *openJPEG* überprüft. Das fehlerhafte Verhalten wird durch die Definition des Zeigers zuvor verursacht, sodass der Schreibprozess durch eine explizite Schleife ersetzt werden musste. Außerdem stellt sich heraus, dass der GCC Compiler die MIPS-Ladefunktionen für die versetzten Speicheradressen fehlerhaft interpretierte. Wenn diese Befehle gemäß der Spezifikation in VHDL modelliert werden, werden die Daten aus einem byte-adressierten Speicher nicht richtig geladen. Der ursprüngliche *plasma*-Kern verzichtete bereits auf diese Modellierung (Tabelle 3.1). Dieser Ansatz wurde auch für die veränderte CPU übernommen. Die Änderungen betrafen keine Kernfunktion des JPEG2000-Standards, sodass von einer nahezu vollständigen Portierung ausgegangen werden kann.

Parallelisierung

Der Komprimierungsalgorithmus des JPEG2000-Standards bietet mehrere Parallelisierungsansätze [JPEG, 2013]. Der erste Ansatz zielt auf die unabhängige Verarbeitung der Farbkoordinaten. In diesem Fall können zwar nur drei parallele Stränge extrahiert werden (dies entspricht der Anzahl der Koordinaten), es besteht allerdings eine hohe Wahrscheinlichkeit, dass die jeweilige Implementierung diese Verarbeitung ebenfalls unabhängig voneinander umsetzt. Eine andere Möglichkeit für eine Parallelisierung setzt tiefer an und nutzt die Tatsache aus, dass der Algorithmus das Bild in gleich große Abschnitte aufteilt und sie dann unabhängig voneinander komprimiert.

Die erste Aufgabe bei der Parallelisierung bestand also darin, die Parallelisierungsansätze im Quellkode zu finden. Es wurde eine Laufzeitanalyse mithilfe der *3DMemory*-Software durchgeführt, um Funktionen zu entdecken, die die Komprimierungsroutinen übernehmen. Die Analyse ergab, dass die Funktion `opj_t1_encode_cblks` 47 % der Ausführungszeit beansprucht. Innerhalb dieser Funktion konnte eine Schleife identifiziert werden, die die Verarbeitung der Farbkoordinaten übernimmt. Daneben wurden noch weitere, in der folgenden Tabelle aufgelisteten Funktionen als Kernfunktionen markiert.

4.5.1 Kernfunktionen der openJPEG-Implementierung während einer Komprimierung, die Dekomprimierungsfunktion wird in Klammern dargestellt.

```
opj_t1_encode_cblks (opj_t1_decode_cblks)
opj_tcd_rateallocate
opj_tcd_makelayer
```

Innerhalb der Hauptschleife konnte die Bearbeitung der einzelnen Bildabschnitte allerdings nicht lokalisiert werden, sodass die Parallelisierung auf **maximal drei Kerne beschränkt ist**.

Die Laufzeitanalyse lieferte außerdem Funktionen, die einen signifikanten Anteil der Laufzeit ausmachten. Die Schleifen innerhalb dieser Funktionen wiesen aber Datenabhängigkeiten auf. Da die Parallellisierung an sich nur als Nebenaufgabe im Rahmen dieser Arbeit einzustufen ist, wurde kein weiterer Aufwand dafür betrieben, Konzepte für die Auflösung dieser Abhängigkeiten umzusetzen.

Die Parallelisierung beschränkte sich darauf, die Schleifendurchläufe auf die Kerne der Testplatform zu verteilen und nach der Schleife Synchronisationspunkte zu setzen. Dieser Ansatz wurde möglichst allgemein gehalten [Ziegler, 2015], um bei anderen Testapplikationen mit der gleichen Methode vorzugehen.

Eingangsdaten

Neben der Applikation selbst stellen die Verarbeitungsdaten eine weitere Variable dar, die das Verhalten des Gesamtsystems beeinflussen kann. Insbesondere ein Komprimierungsprozess hängt stark davon ab, welche Variabilität die Rohdaten aufweisen [Strutz, 2005]. Die Software *openJPEG* akzeptiert eine Reihe an Bildformaten als Eingabe. Die Einfügefunktion wandelt die Daten dabei in ein internes Format um.

Daher ist es für die Kernfunktion der Applikation nebensächlich, in welcher Form die Eingangsdaten vorliegen. Für diese Arbeit wurde das Format **bitmap** von Microsoft [Microsoft, 2014] benutzt.

Neben der Vielfalt an Eingangsformaten stellt die Software gemäß JPEG2000-Spezifikation auch eine Reihe an Kodierungseinstellungen zur Verfügung, die den Verarbeitungsprozess maßgeblich beeinflussen können. Die Variation dieser Einstellungen bietet zwar einen weiteren Ansatzpunkt für eine Speichernutzungsanalyse, wurde im Rahmen dieser Arbeit aber nicht weiterverfolgt. Es wurde angenommen, dass die Ergebnisse eher für Entwickler von Bildverarbeitungsalgorithmen von Interesse sind. Stattdessen wurden die Komprimierungseinstellungen auf Standardwerte der Software mittels Funktionen `opj_set_default_encoder_parameter` für Komprimierung und `opj_set_default_decoder_parameter` für Dekomprimierung gesetzt. Der einzige Parameter, der von der Bildgröße abhängig ist, ist `numresolution`. Sein Wert hängt von der längsten Kante des Bildes ab und entspricht dem Wert des Zweierlogarithmuse dieser Länge. Daher wurde der Wert des Parameters nach dem Bildimport entsprechend angepasst. Der Wert dieses Parameters beeinflusst den Programmablauf geringfügig. Daher unterscheiden sich die Bearbeitungsprozesse von kleinen und großen Bildern.

Die verwendeten Bilder sind der Abbildung 4.7 zu entnehmen.

Abbildung 4.7: Verwendete Bilder [USC Viterbi School of Engineering, 2013] für Analyse der JPEG2000 Implementierung

Die Größe der Bilder wurde für die Untersuchung variiert, wobei die quadratische Form beibehalten wurde. Im Anhang A.1 ist eine maßstabsgetreue Abbildung der Größenverhältnisse zu finden.

4.5.2 BZIP2 – allgemeine Datenkompression

Algorithmus und Implementierung

Im Gegensatz zu JPEG2000 implementiert das bzip2-Programm keinen abgeschlossenen Standard. Die von Julian Seward entwickelte Software basiert auf Burrows-Wheeler-Transformation und Huffman-Kodierung. Die Applikation nimmt beliebige Eingangsdaten entgegen und reduziert ihre Größe. Die Komprimierungsstufe und damit die Rechenzeit sind einstellbar, wobei die Komprimierung verlustfrei erfolgt [Julian Seward, 2015]. Daher ist die Größenreduktion ebenfalls einstellbar, allerdings bis zu einer Grenze, die die informationstheoretische Entropie der Quelle vorgibt.

Die Software ist quelloffen und frei von patentierten Implementierungstechniken.

Portierung

Die Übersetzung des bzip2-Programms durch den GCC-Compiler ergab, dass die Programmausführung von einer Reihe an Aufrufen externer C-Bibliotheksfunktionen begleitetet wurde. Die Aufgaben dieser Funktionen umfassen nahezu ausschließlich die Behandlung von Fehlerfällen beziehungsweise Statusausgaben. Da diese Funktionalität für den eigentlichen Komprimierungsprozess nicht notwendig ist, wurden die Funktionen aus dem Programmfluss entfernt. Eine Simulation der Ausführung ergab die gleichen Ausgabedaten wie ein x86-Referenzsystem [Dahlem, 2015].

Parallelisierung

Das bzip2-Programm verfügt bereits über einen Ableger, der die Implementierung parallelisiert: [Jeff Gilchrist, 2015]. Der Aufbau des Komprimierungsvorgangs liefert direkt einen Parallelisierungsansatz. Die Eingangsdatei wird in gleich große Abschnitte unterteilt, die unabhängig voneinander komprimiert werden. Beim ungekehrten Prozess werden die Abschnitte ebenfalls unabhängig voneinander wiederhergestellt. Daher führt ein Fehler innerhalb eines Abschnitts im ungünstigsten Fall dazu, dass nur dieser Abschnitt verworfen wird, ohne die komplette Datei unbrauchbar zu machen [Julian Seward, 2015]. Gleichzeitig kann die Verarbeitung dieser Abschnitte auf mehrere Kerne verteilt werden.

In dieser Arbeit wurde der Ansatz von PBzip2 auf das Netzwerk des Mehrkernsystems übertragen. Dabei konnte das allgemeine Verteilungskonzept, das bei openJPEG entwickelt wurde, verwendet werden [Dahlem, 2015].

Eingangsdaten

Das Programm ist in der Lage, beliebige Formate als Eingangsdaten zu verarbeiten, da die Leseroutine bei der binären Repräsentation ansetzt. Die Kategorie des Dateieninhalts ist für das Programm somit nicht relevant. Die Suche nach einer Zusammenstellung von häufig verwendeten Dateiformaten und -inhalten für Kompressionssoftware blieb leider erfolglos. Daher wurden willkürlich folgende Eingangsdaten verwendet [Dahlem, 2015]:

- Eine Fassung des Theaterstücks *Romeo und Julia* im DOC-Format von Microsoft.
- Teile des Quellkodes des Linux-Kernels
- Ein selbsterstelltes Bild, um einen Bezug zu JPEG2000 herzustellen.

Der einzige Komprimierungsparameter, der angepasst wurde, ist die Blockgröße. Diese Größe war für die Parallelisierung relevant. Der Wert wurde auf das Minimum von 100 Kilobyte gesetzt, um den größtmöglichen Effekt durch die Parallelisierung beobachten zu können.

Die Größe der Eingangsdaten ging als eine Variable in die Untersuchungen ein. Die Größenordnung für die konkreten Werte der Dateiengrößen orientierte sich an den Ergebnissen der Untersuchung zu JPEG2000.

4.5.3 MPEG-4 - Videokompression

Algorithmus

Eine Spezifikation der Video- und Audiokompression wird unter dem MPEG-Standard erstellt. Die 1988 gegründete Expertenrunde fasst in der vierten Version des Standards eine Reihe von Vorgaben zusammen, um die Behandlung von Audio- und Videodateien portabel zu gestalten [MPEG, 2015].

Der aktuelle Standard umfasst eine breite Anwendungspalette. Für diese Arbeit waren nur die Abschnitte zur Videokomprimierung von Bedeutung. Der Standard definiert zwei solcher Abschnitte (Teil 2 und Teil 10). Implementierungen dieser beiden Abschnitte wurden als Testapplikationen ausgewertet.

XviD – Implementierung des 2. Teils

Das XviD-Programm implementiert den zweiten Teil des Standards und führt die Kernberechnungen ohne den Einsatz von Fließkommazahlen durch [Xvid, 2010]. Die Software hat ihren Ursprung im MoMuSys-Projekt [MoMuSys, 1997] und wird von einigen Programmierern freiwillig weiterentwickelt. Der Quellkode unterliegt der GPL-Lizenz.

Die Software ist in der Programmiersprache C implementiert. Bei der Portierung musste nur der Compiler den Vorgaben entsprechend konfiguriert werden. Die Ausgabedatei weicht geringfügig von den Referenzausgaben eines x86-Rechners ab. Dies betrifft allerdings nur optionale Felder im Kopf der Datei [Dahlem, 2015].

Die Implementierung bietet zwei Parallelisierungsansätze, die bereits Teil des Quellkodes sind. Der erste Ansatz greift tief in die Berechnung ein und verteilt bestimmte Funktionen der Komprimierungsroutine. Die zweite Methode setzt bei den Eingangsdaten an und nutzt die Tatsache aus, dass bei der Videokomprimierung unterschiedliche Bildformate existieren. Ein Format (das *I-Frame*) enthält vollständige Informationen über den Inhalt, die anderen Formate übertragen entweder Änderungen (*B-Frame*) oder Vorhersagen der Änderungen (*P-Frame*) [Strutz, 2005]. Das *I-Frame* enthält Daten, die unabhängig von vorhergehenden Übertragungen sind. Daher kann eine Bildreihe, die mit diesem Format beginnt, parallel erstellt werden. Beide Ansätze wurden für diese Arbeit genutzt [Dahlem, 2015].

Im Vergleich zu einem Bild bietet ein Video eine größere Auswahl an Inhaltsbestandteilen, die die Komprimierung beeinflussen könnten. Für diese Arbeit wurden drei Videos ausgewählt, die sich jeweils durch eine bestimmte Eigenschaft charakterisiert lassen, wie die Abbildung 4.8 zeigt.

(1) (2) (3)

Abbildung 4.8: Die ersten Frames der Originalvideos, die für die Analyse des XviD-Programms verwendet wurden [Dahlem, 2015]. (1) zeigt eine Aufnahme, die von einer unbeweglichen Kamera erfasst wurde. (2) zeigt ein Video, dessen Farbspektrum eingeschränkt ist. (3) ist ein Handy-Video, das typische Verwackelungen aufweist.

Eine Variation der Dateigröße kann bei Videos auf zwei Wegen erreicht werden: durch die Änderung der Framegröße oder durch die Änderung der Länge des Videos. Bei dieser Arbeit wurde die Größe auf 64 mal 64 Bildpunkte festgelegt, die Videos also entsprechend skaliert. Anschließend wurde die Laufzeit der Videos soweit variiert, dass vergleichbare Dateigrößen erreicht wurden, wie bereits bei JPEG2000 und bzip2.

x.264 – Implementierung des 10. Teils

Der 10. Teil des MPEG-IV-Standards beschreibt einen Komprimierungsalgorithmus, der für die gleiche Dateigröße eine „bessere“ Bildqualität erreichen kann als der Algorithmus aus dem 2. Teil des Standards [MPEG, 2015]. Eine Implementierung ist das gleichnamige x.264-Programm [x264, 2015]. Die Berechnung stützt sich dabei nahezu ausschließlich auf Operationen mit Fließkommazahlen. Der verwendete Prozessorkern bietet in seiner Originalversion keine Hardware-Unterstützung für diese Operationen 3.1.1. Die Berechnungen werden durch entsprechende Softwareroutinen emuliert, sodass die Rechenzeit signifikant zunimmt. Darum wurde der veränderte Prozessorkern durch eine FPU-Komponente ergänzt 3.1.3.

Der Einsatz dieser FPU offenbarte jedoch schwerwiegende Probleme. Die Auslagerung der Fließkommaoperationen über eine externe Schnittstelle des Simulators auf die Hardware des Arbeitsservers führte zu einem Abbruch der Simulation, da während der Berechnung Randbereiche der Darstellung der Fließkommazahlen erreicht wurden. Die Behandlung dieser Randbereiche war vermutlich bei der verwendeten Konstellation nicht vorgesehen, sodass die Ausführung unmittelbar unterbrochen wurde. Externe VHDL-Bibliotheken für Fließkommaoperationen konnten zwar mit diesen Randbereichen umgehen, die reale Simulationszeit konnte durch ihren Einsatz jedoch nicht signifikant verringert werden [Rausch, 2015]. Der simulative Einsatz einer FPU-Komponente wurde damit aus Zeitgründen nicht weiterverfolgt.

Eine Ausführung des portierten x.264-Programms lieferte fehlerhafte Daten. Die Fehlersuche wurde durch eine lange Simulationszeit von zweieinhalb Stunden erschwert. Die meiste Zeit davon führt das Programm Initialisierungsberechnungen durch, die abhängig vom Eingangsvideo sind. Diese Phase ist unabhängig von der Größe des Videos, sodass der bis dahin übliche Weg der Reduktion von Eingangsdaten die Simulationszeit nicht reduzieren konnte [Rausch, 2015]. Diese Tatsache führte zum Entschluss, das Programm aus der Reihe der Testapplikationen zu entfernen, da eine Analyse mit einem DRAM-Model den vertretbaren Zeitrahmen gesprengt hätte.

4.6 Synthetische Funktionen

Der zentrale Bestandteil der Hypothesen 2 und 3 dieser Arbeit ist die Pufferspeicherfunktionalität. Diese Funktionalität basiert auf der Lokalitätseigenschaft. Die zuvor vorgestellten Applikationen stellen real anwendbare Fälle dar. Es stellt sich allerdings die Frage, wie diese Applikationen in Bezug auf die Lokalität einzustufen sind. Eine Grundlage für diese Einstufung können zwei Extremfälle bieten: ein besonders gute und als Gegenpol eine besonders schlechte Lokalität. Insbesondere aus dem zweiten Fall lässt sich ableiten, wie viel Leistungssteigerungspotenzial durch eine Verbesserung des DRAM-Speichers erreicht werden kann.

Streng genommen kann bei einer Applikation zwischen zwei Lokalitäten unterschieden werden. Eine Verteilung der Zugriffe auf Instruktionsadressen beschreibt die häufig genutzten Befehle und eine entsprechende Verteilung der Datenadressen die häufig genutzten Datenstrukturen. Bezüglich der Instruktionen zeigt die Praxiserfahrung, dass die „10-90 %“-Regel ziemlich genau eingehalten wird [Hennessy und Patterson, 2012]. Die Verteilung der Datenlokalität ist dagegen stärkeren Schwankungen unterworfen. Außerdem ist es in der Regel die Datenmenge, die in der Lage ist, die Kapazität des Pufferspeichers zu überschreiten. Daher konzentrieren sich die folgenden Funktionen ausschließlich auf diese Lokalität.

4.6.1 Gute Datenlokalität

Die Lokalitätseigenschaft des Speichers besagt, dass die meiste Zeit (90 %) nur wenige Daten benutzt werden (10 %). Wenn diese Menge komplett in den Pufferspeicher passt, muss der DRAM nur einmal pro

Datenpaket (*cache line*) gelesen werden. Für die restliche Verarbeitungszeit wird er nicht mehr gebraucht. Anschließend müssen die veränderten Daten allerdings zurück in den DRAM geschrieben werden. Eine Reduktion der DRAM-Nutzung auf diese zwei Zugriffe stellt einen möglichen Extremfall der Speichernutzung dar. Sie diente als Grundlage für die Implementierung der folgenden Funktion (Ausschnitt 4.6.1). Einige Statusausgaben sowie Sicherheitsabfragen in Bezug auf Datei- und Speicherbereichsende wurden aus Einfachheitsgründen im Ausschnitt weggelassen.

4.6.1 Implementierung einer guten Datenlokalität

```
1   unsigned int len, idx, a;
2
3   FILE * IN  = fopen( "INPUT",  "rb" );
4   FILE * OUT = fopen( "OUTPUT", "wb" );
5
6   fread( &len, sizeof(unsigned int), 1, IN );
7
8   for( idx = 0; idx < len; idx++ ){
9     fread(  &a, sizeof(unsigned int), 1, IN );
10    fwrite( &a, sizeof(unsigned int), 1, OUT );
11  }
```

Die Funktion liest den ersten Eintrag in einer Datei, interpretiert ihn als Größenangabe und initiiert anschließend in der Hauptschleife eine entsprechende Anzahl an Lese- und Schreibzugriffen. Durch die Angabe der Größe konnten unterschiedliche Mengen an Eingangsdaten emuliert werden, um mit den Eingangsdaten der realen Applikationen vergleichbar zu bleiben. Die Funktion ist nicht explizit darauf ausgelegt, möglichst viele Speicherzugriffe im Pufferspeicher verorten zu lassen. Wenn die Größe des Pufferspeichers es allerdings zulässt, alle Daten zu behalten, liegt das eingangs beschriebene Szenario der minimalen Anzahl an DRAM-Zugriffen vor.

4.6.2 Schlechte Datenlokalität

Eine schlechte oder gar nicht vorhandene Lokalität liegt vor, wenn die Verteilung der Adressen zufällig ist. Die einzige Aufgabe der im Ausschnitt 4.6.2 dargestellten Funktion bestand also darin, diesen Zufall zu generieren.

So sehr der Zufall Teil der Physik ist, so unerwartet kompliziert ist es, eine Erzeugung von echten Zufallszahlen mittels digitaler Komponenten zu realisieren. Diese Aufgabe markiert sogar einen eigenen Forschungsgegenstand. Für diese Arbeit wurde ein Pseudozufallszahlengenerator von Matsumoto und Nishimura [Matsumoto und Nishimura, 1998] in der Version „MT19937 “ benutzt. Der Aufbau und vor allem der mathematische Hintergrund des Generators wurden im Rahmen dieser Arbeit nicht weiter vertieft. Der relevante Teil seiner Funktionen wurde direkt übernommen.

4.6.2 Implementierung einer schlechten Datenlokalität

```
1  unsigned int  len, idx, data;
2  unsigned long addr;
3  unsigned long init[4]={0x123, 0x234, 0x345, 0x456}, length=4;
4
5  init_by_array(init, length);
6
7  FILE * IN = fopen( "INPUT", "rb" );
8
9  fread( &len, sizeof(unsigned int), 1, IN );
10
11 for( idx = 0; idx < len; idx++ ){
12   addr = genrand_int32();
13
14   data = *((unsigned int *)addr);
15   *((unsigned int *)MEM_OUTD_START) = data;
16 }
```

Die Funktion beschränkt sich ausschließlich auf den Lesevorgang. Die Adresse wird dabei zufällig generiert. Der Schreibvorgang diente in erster Linie dazu, den Compiler davon abzuhalten, den Lesevorgang als ungenutzt wegzuoptimieren. Wie schon bei der vorigen Funktion kann die Anzahl der Lesezugriffe mit dem ersten Eintrag der (sonst unbenutzten) Eingangsdatei angegeben werden. Die zufällige Verteilung von Lesezugriffen soll permanente Ersetzungsprozesse im Pufferspeicher provozieren und seinen Einsatz dadurch praktisch mit einer zusätzlichen Verzögerung gleichsetzen. Im ungünstigsten Fall erzeugt jeder Lesezugriff einen DRAM-Datentransfer.

5 Untersuchungen auf einem Einkernsystem

5.1 DRAM-Anteil an der Laufzeit

Die Ausführung einer Applikation ist ein dynamischer Prozess. Wie jeder Prozess lässt er sich in bestimmte Zeitabschnitte unterteilen. Daher kann nicht von *der* Laufzeit einer Applikation gesprochen werden. Der Abschnitt 2.3 geht vertieft auf diese Problematik ein und definiert das für diese Arbeit verwendete **Messobjekt**. in Bezug auf den **Zeitabschnitt** werden allerdings nur die relevanten Fragestellungen erläutert. Für die konkrete Messung mussten diese Fragen beantwortet werden, wobei zu berücksichtigen ist, dass die Wahl der Antworten die Messergebnisse beeinflussen kann. Bei allen Untersuchungen wurde der Speicher vor Beginn der Messung vollständig mit Instruktionen und Eingangsdaten geladen. Die Initialisierungsphase des DDR2-DRAM wurde aus der Messung herausgenommen. Die Bearbeitungsparameter der Applikationen sowie die Eingangsdaten wurden bereits im vorigen Kapitel beschrieben. Nach der Ausführung musste mit Pufferspeicher - bei Systemen, die einen Pufferspeicher beinhalten - die Daten in den DRAM übertragen. Diese Übertragungszeit wurde ebenfalls aus der Messung herausgenommen.

Eine Komponente des Messobjekts stellen die Eingangsdaten der Applikationen dar. Der Inhalt dieser Daten spielt nur für reale Applikationen eine Rolle, daher wurden auch drei unterschiedliche Inhalte je Applikation getestet. Es stellt sich allerdings in diesem Zusammenhang die Frage, wie die Messergebnisse dargestellt werden sollen, wenn der Inhalt der Eingangsdaten einen Einfluss auf die Laufzeit der Applikation hat. Die erste Messung dieser Arbeit lieferte die Anzahl der Berechnungszyklen, die eine Applikation für die Ausführung braucht. Der Speicher wurde durch eine ideale Komponente modelliert, sodass er keinen Einfluss auf die Berechnungszeit hatte. Doch bevor die Ergebnisse vorgestellt werden, sei ein Blick auf die Streuung der Zyklenzahl - dargestellt als Standardabweichung - in Abhängigkeit von der Größe der Eingangsdaten erlaubt (Tabelle 5.1).

Tabelle 5.1: Standardabweichungen, bezogen auf die Anzahl der Verarbeitungszyklen für unterschiedliche Eingangsdaten

	20 kB	60 kB	200 kB	1000 kB
openJPEG	6,62 %	5.84 %	5,31 %	7,27 %
bzip2	78,79 %	13,63 %	3,15 %	8,32 %
XviD	11,12 %	11,82 %	14,32 %	16,49 %

Die Zahlen lassen erkennen, dass die Eingangsdaten einen nicht zu vernachlässigenden Einfluss auf die Bearbeitungszeit haben, auch wenn die Größe der Daten konstant gehalten wird. Insbesondere für das Programmm bzip2 lässt sich eine große Streuung für die kleinste Datei messen. Diese Werte lassen sich auf die Komprimierung der Bilddatei zurückführen. Diese Datei beinhaltet bei der kleinsten Größe beinahe keinen variablen Inhalt mehr. Die Entropie ist sehr gering. Bis auf einige Farbpunkte ist der Rest weiß. Das bzip2-Programm scheint diesen Mangel an Entropie zu erkennen und die Verarbeitung entsprechend zu verkürzen. Ansonsten ist keine allgemeine Entwicklung in Abhängigkeit von der Eingangsgröße zu erkennen. Die Werte scheinen willkürlich zu schwanken. Basierend auf diesen Erkenntnissen werden in den nächsten Diagrammen neben dem Mittelwert auch der maximale und der minimale Wert abgebildet.

Auf der nächsten Seite ist die absolute Anzahl an Takten, die für eine Verarbeitung benötigt wurden, für jede Testapplikation dargestellt (Abbildung 5.1). Neben einer zusätzlichen Linie, die die maximalen und

minimalen Werte markiert, ist zusätzlich der mittlere Betrag abgebildet (die Y-Achse hat einen logarithmischen Maßstab). Das Wachstum der Zykluszahl erfolgt exponentiell und ist annähernd proportional zum Größenwachstum der Eingangsdaten. Die synthetischen Funktionen benötigen dabei weniger Takte als reale Applikationen, wobei die Bildkompression mittels openJPEG die größte Anzahl an Instruktionen aufruft. Diese Tatsache scheint auf den ersten Blick zu überraschen, da die Videokompression eine ähnliche Aufgabe übernimmt und zusätzlich die Bildveränderungen einrechnet. Eine mögliche Erklärung für diese Messwerte könnte die Betrachtung der Auflösung der jeweiligen Bilder liefern. Während für openJPEG das Wachstum der Eingangsdaten durch Zunahme der Bildpunkte pro Kantenlänge erreicht wurde, blieb dieser Wert für getestete Videos konstant. Nur die Anzahl der Frames trug zu größeren Dateien bei.

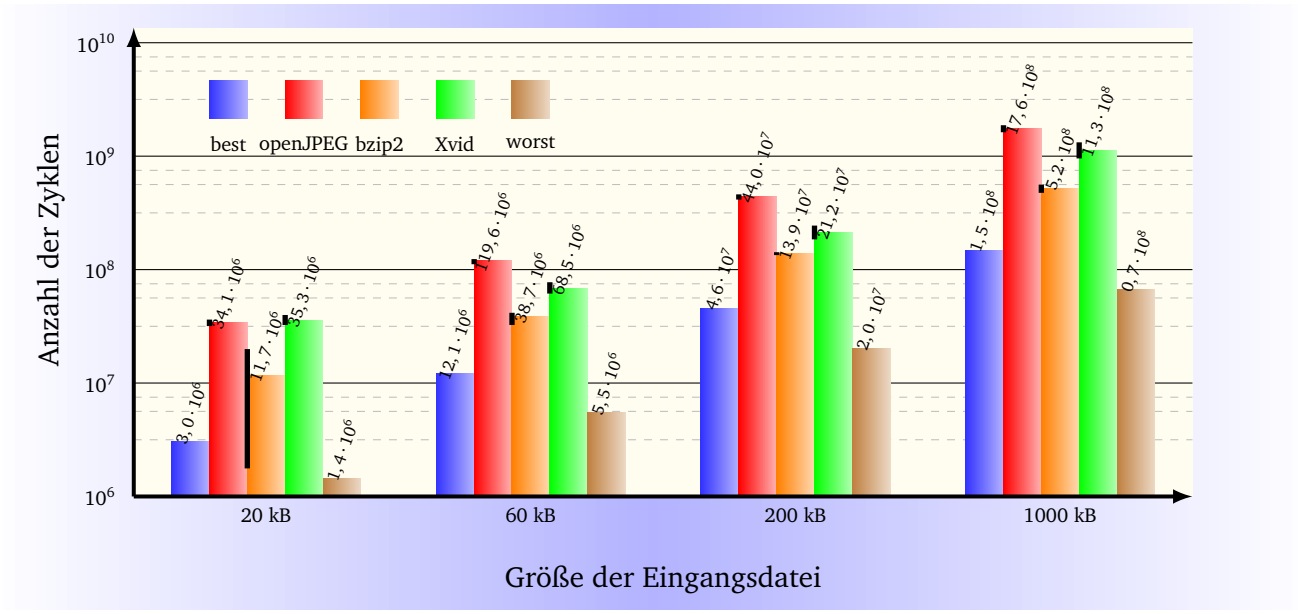


Abbildung 5.1: Anzahl der Taktzyklen für Datenverarbeitung in Abhängigkeit von der Größe der Eingangsdaten. Die Y-Achse hat einen logarithmischen Maßstab, um die exponentiell wachsenden Werte besser darzustellen.

An dieser Stelle wird deutliche, dass es mehrere Möglichkeiten gibt, den Inhalt einer Testdatei zu variieren, um verschiedenen Größen dieser Datei zu bekommen. Abhängig vom gewählten Parameter kann die Applikation verschiedene Verarbeitungspfade einschlagen und somit zu anderen Ergebnissen führen. Bei der Feststellung des Einflusses des Speichers sollte dieser Effekt möglichst herausgerechnet werden. Mit der Kenntnis der Zyklenzahl lässt sich die Laufzeit für beliebige Betriebsfrequenzen berechnen. Die in dieser Arbeit untersuchte Größe ist daher die **relative Verzögerung**, deren Berechnung in der folgenden Gleichung dargestellt ist:

$$\text{Relative Verzögerung} = \overbrace{\underbrace{\frac{\text{Laufzeit} - \text{Zyklenzahl} * \text{Taktzeit}}{\text{Zyklenzahl} * \text{Taktzeit}}}_{\textit{ideale Laufzeit}}}^{\textit{absolute Verzögerung}}$$

Die absolute Verzögerung lässt sich aus der Differenz zwischen der gemessenen Laufzeit und der idealen Laufzeit basierend auf der Anzahl der Zyklen berechnen. Der Wert dieser Verzögerung kann allerdings nur in Bezug auf die komplette Ausführungszeit eingeordnet werden. Aus dem Wert für die relative Verzögerung lässt sich ablesen, inwieweit das Speichersystem die Ausführungszeit verlängert und welchen Gewinn eventuelle Optimierungsmaßnahmen bringen können.

Im Abschnitt 3.3.1 wird das Speicherreferenzmodell vorgestellt. In Abbildung 3.8 wird verdeutlicht, wie der Einfluss des Pufferspeichers aus dem Messwert herausgerechnet wird. Die Architektur des Pufferspeichers hat zwar einen unmittelbaren Einfluss darauf, wie oft der DRAM angesprochen wird, Verzögerungen innerhalb der Ebenen des Pufferspeichers dürfen aber nicht in die Beurteilung der Größe der DRAM-verursachten Verzögerung einfließen. Daher galt es zunächst, den Anteil des Pufferspeichers an der gesamten Verzögerung zu ermitteln. Der DRAM wurde dabei durch eine ideale Komponente ersetzt. Abbildung 5.2 zeigt die gemessenen Werte für die relativen Verzögerungen.

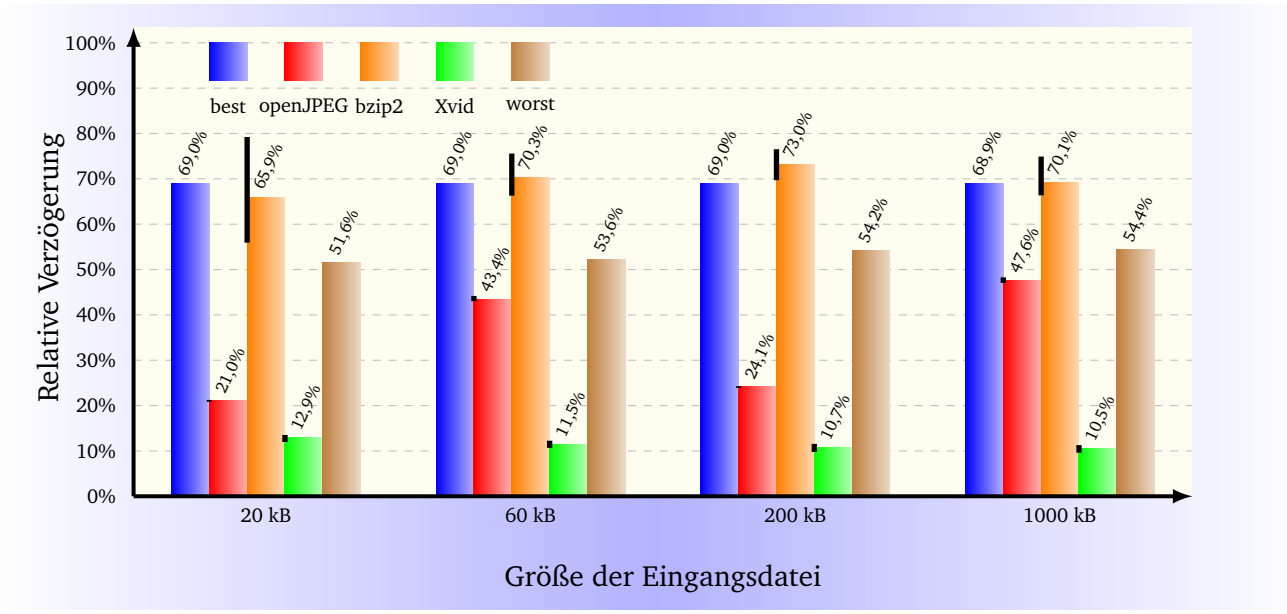


Abbildung 5.2: Relative Verzögerung durch den Pufferspeicher in Abhängigkeit von der Größe der Eingangsdatei.

Auffällig ist eine Unabhängigkeit der Werte von der Größe der Eingangsdaten. Sowohl für kleine als auch für große Daten wird im Verhältnis zur Zykluszahl die gleiche Anzahl an Übertragungen innerhalb der Pufferspeicherebenen ausgeführt. Die Variation ist bis auf das bzip2-Programm relativ gering. Die Markierungslinien für maximale und minimale Werte sind in der Nähe des Mittelwerts zu finden. Das bzip2-Programm nimmt in zweierlei Sicht eine Sonderstellung ein. Auf der einen Seite scheint der Inhalt der Testdateien messbare Auswirkungen auf die Anzahl der Fehlzugriffe innerhalb des Pufferspeichers zu haben. Diese Beobachtung lässt vermuten, dass das Programm eine geringe Datenlokalität besitzt. Auf der anderen Seite ist die relative Verzögerung, die durch den Pufferspeicher verursacht wird, bis auf die erste Testreihe am größten. Die verwendete Pufferspeicherkonfiguration ist somit nur bedingt in der Lage, dieses Programm zu beschleunigen.

Die synthetischen Programme zeigen ebenfalls eine hohe Verzögerung durch den Pufferspeicher, wobei im Falle der guten Datenlokalität der hohe Wert überrascht. Die Werte für Bild- und Videokompressionen sind stets kleiner. Bei openJPEG scheinen bestimmte Eingangsdaten den Pufferspeicher besser auszunutzen. Die Werte für 200 kB sind kleiner als die für 60 kB.

Mit der Kenntnis des Pufferspeicheranteils lässt sich der DRAM-Anteil wie folgt berechnen:

DRAM-Anteil = relative Verzögerung - relative Verzögerung (Pufferspeicher)

Wenn nichts anderes angegeben ist, wird im Folgenden nur der DRAM-Anteil betrachtet.

5.2 Vergleich zwischen DDR2- und 3D-DRAM

5.2.1 Vergleich für reale Applikationen

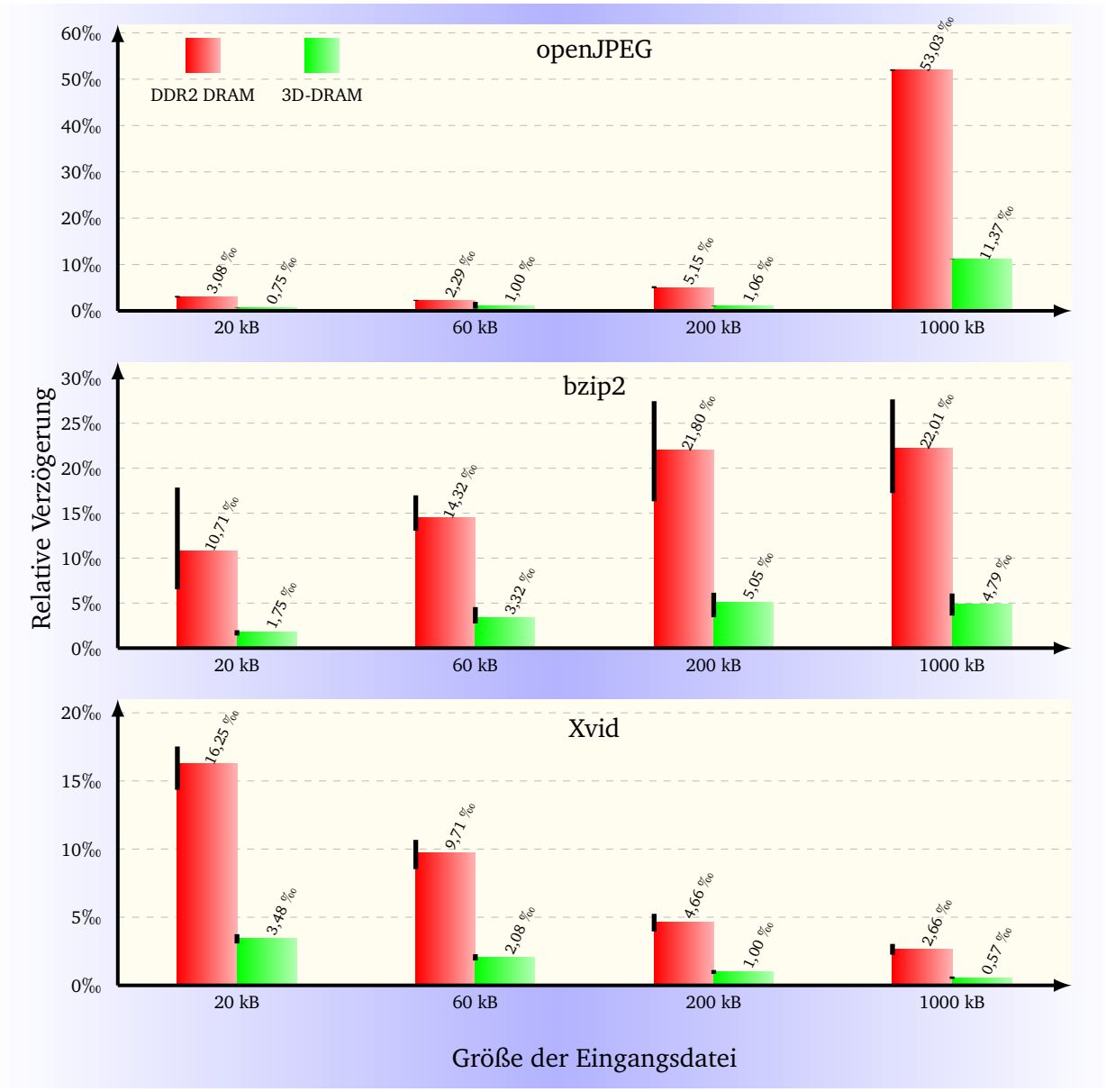


Abbildung 5.3: Vergleich der relativen Verzögerung zwischen einem System mit DDR2-DRAM und 3D-DRAM in Abhängigkeit von der Applikation und der Größe der Eingangsdatei [Schoenberger und Hofmann, 2014b]

Die Betrachtung der aktuellen Forschungsarbeiten (Abschnitt 2.2.2) hat gezeigt, dass die Stapeltechnik eine Reihe an Möglichkeiten bietet, die Leistungsfähigkeit eines 3D-DRAM zu steigern. Das für diese

Arbeit verwendete Modell wurde im Abschnitt 3.3.3 vorgestellt und lässt sich im Wesentlichen auf eine kürzere Zugriffszeit zurückführen. Eine Reduktion dieser Zeit sollte einen messbaren Effekt bei der relativen Verzögerung hervorrufen. Abbildung 5.3 zeigt die gemessenen Werte für ein System mit DDR2-DRAM im Vergleich zu einem System mit 3D-DRAM in Abhängigkeit von der Größe der Eingangsdatei und der Applikation. Der Controller für 3D-DRAM bot eine Funktionalität, die es erlaubte die Befehlsfolge bei einem *write-back*-Zugriff zu vertauschen, um die Haltezeit der CPU zu verringern (Abschnitt 3.3.3). Diese Optimierungsstufe wurde nur bei diesem Vergleich aktiviert. In allen weiteren Untersuchungen wurde diese Funktion nicht verwendet, da sie dort bei einigen Testdurchläufen zu Fehlern führte.

Der erwartete Effekt lässt sich klar erkennen:

> Für jede Applikation und jede Eingangsdatei ist die relative Verzögerung durch 3D-DRAM kleiner als durch DDR2-DRAM.

Bis auf diese Feststellung lassen sich zunächst keine weiteren Gemeinsamkeiten zwischen den Testfällen beobachten. Bei openJPEG ist der DRAM-Anteil bis auf die größte Datei gering. Genauso ist es für ein System mit DDR2. An dieser Stelle sei noch angemerkt, dass die Ausführung für die größte Datei bei openJPEG und DDR2 mit einem Fehler für alle drei Bilder abgebrochen wurde. Die Fehlermeldung kam vom Speichermodel und verwies auf Verletzungen von Zugriffszeiten. Die Einhaltung von Zugriffszeiten übernahm bei dieser Konstellation ein automatisch generierter DRAM-Controller von Xilinx. Zudem erschwerte die Menge der Simulationsdaten eine Fehlersuche enorm. Daher wurde bei openJPEG nur für diesen Vergleich nicht die komplette Laufzeit ausgewertet. Das Programm ist relativ stark in Unterprogrammme gegliedert, sodass der Verarbeitungsprozess während der Ausführung nachverfolgt werden kann. Die Werte aus der Ausführung auf dem System mit 3D-DRAM wurden entsprechend angepasst, sodass für alle Bilder der gleiche Verarbeitungsstand als Messgrundlage benutzt wurde.

Das bzip2-Programm zeigt eine Zunahme der relativen Verzögerung für größere Dateien. Die Werte sind im Vergleich zu openJPEG bis auf den letzten Durchgang größer. Weiterhin ist eine große Streuung der Messwerte für diese Applikation festzustellen, die für alle Durchläufe auffällig signifikant bleibt. Wie bereits bei der Betrachtung des Pufferspeicheranteils deutet dieser Befund auf eine geringere Datenlokalität hin, die zudem stark vom Inhalt der Eingangsdaten abhängt.

Das XviD-Programm zeigt eine umgekehrte Entwicklung. Je größer die Eingangsdaten werden, desto geringer wird der DRAM-Anteil. Die Streuung der Messwerte ist zwar größer als bei openJPEG, die Mittelwerte scheinen sich aber ab 1000 kB Daten den kleinen Werten von openJPEG bei 20 kB anzunähern. Diese Entwicklung deutet darauf hin, dass die XviD-Applikation eine besonders gute Datenlokalität besitzt.

5.2.2 Vergleich mit synthetischen Funktionen und Einordnung der Applikationen

Eine Anordnung der Applikationen in Bezug auf die Datenlokalität erlauben die synthetischen Funktionen, die jeweils die beiden Extremverhalten nachbilden sollen. Die Abbildung 5.4 zeigt zunächst den Vergleich zwischen den DRAM-Systemen. Bei einer guten Datenlokalität muss der DRAM-Anteil für größere Eingangsdaten abnehmen, da die absoluten Werte für DRAM-Zugriffe in Relation zur Laufzeit kleiner werden. Die Daten sollten sich im Pufferspeicher befinden. Genau dieses Verhalten lässt sich auf der oberen Grafik in der Abbildung erkennen. Wie bereits bei realen Applikationen ist die relative Verzögerung für ein 3D-DRAM-System systematisch kleiner. Ein System mit einer schlechten Datenlokalität dagegen sollte unabhängig von der Größe der Eingangsdatei einen gleich (hohen) Wert haben, da die Daten per Zufall ausgewählt werden. Die Messung zeigt, dass der DRAM-Anteil in diesem Fall stabil ca. 50 % der Laufzeit ausmacht.

Ausgehend von diesen Messungen sind in der Abbildung 5.5 die Werte für ein DDR2-DRAM-System und 1000-kB-Eingangsdaten nochmals im Applikationsquervergleich abgebildet. Die X-Achse der Grafik

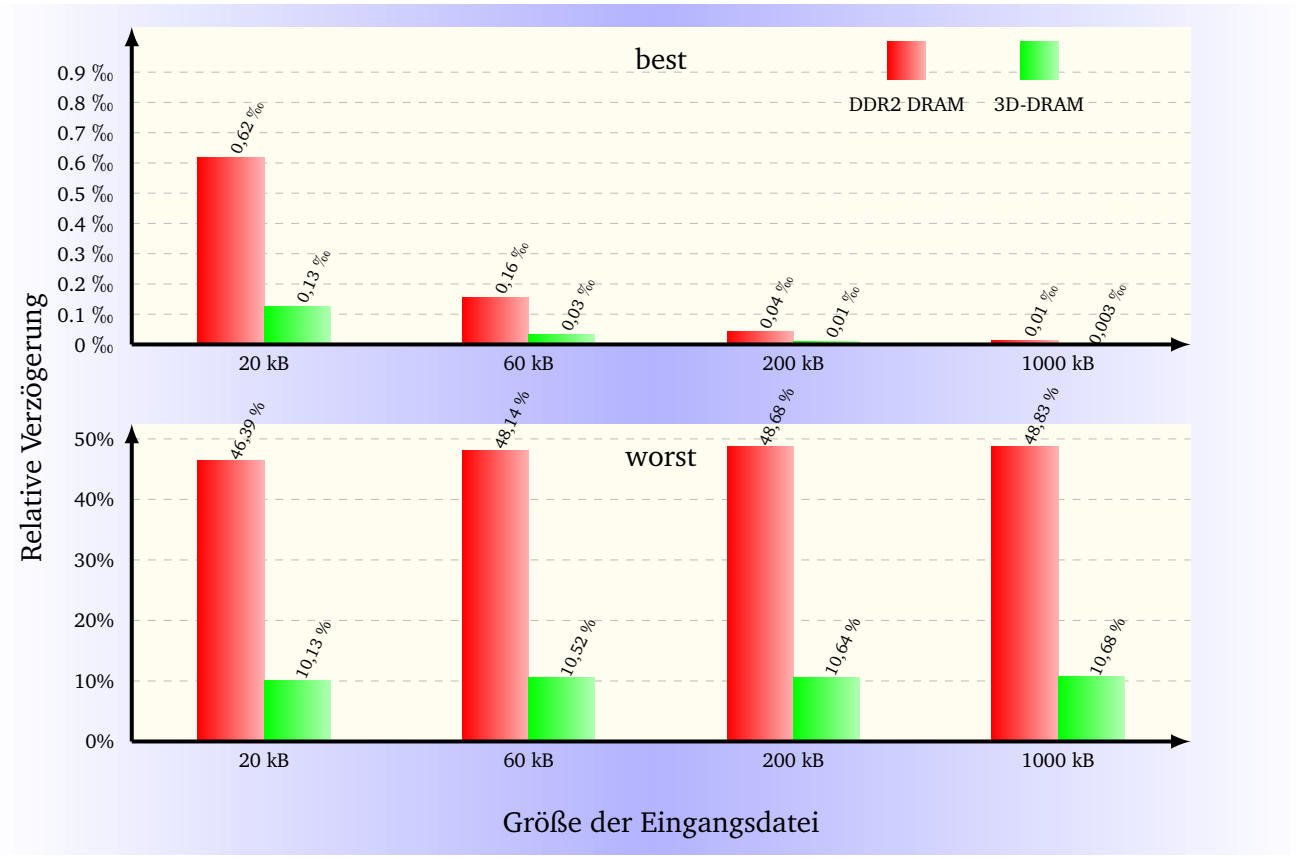


Abbildung 5.4: Vergleich der relativen Verzögerung zwischen einem System mit DDR2-DRAM und einem System mit 3D-DRAM für Funktionen mit guter und schlechter Datenlokalität.

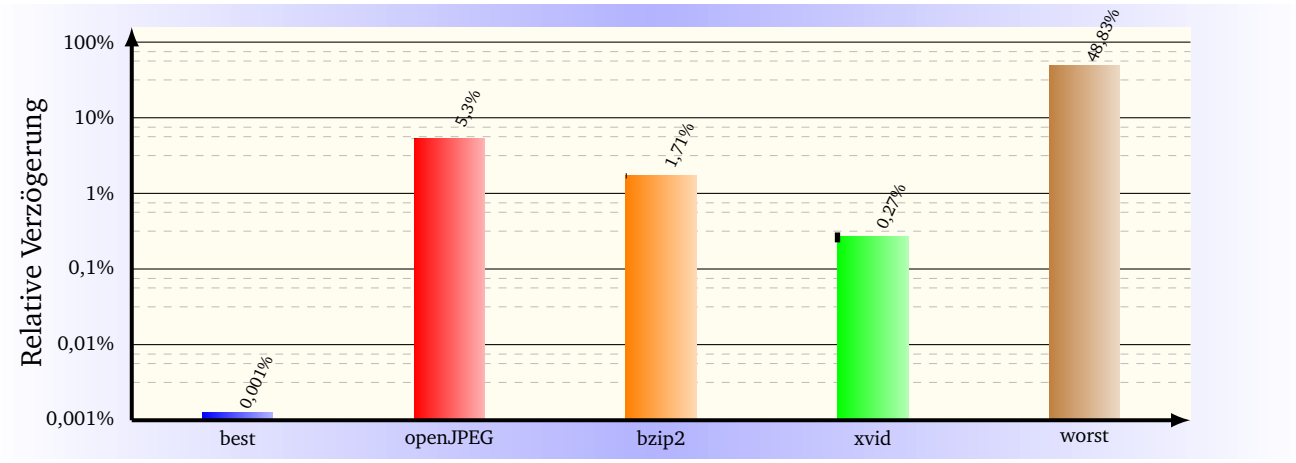


Abbildung 5.5: Einordnung der realen Applikationen in Bezug auf die Datenlokalität

hat einen logarithmischen Maßstab, um die Werte bei sehr guter Datenlokalität noch zeichnen zu können. Die zuvor vorgestellten Werte zeigten bereits, dass die relative Verzögerung sowohl von der Größe der Eingangsdaten als auch von deren Inhalt abhängt. Daher basiert die folgende Feststellung nur auf den betrachteten Werten:

Die untersuchten Applikationen weisen für die relative Verzögerung durch den DRAM einen Wert, der tendenziell auf eine schlechte als auf ein gute Datenlokalität hindeutet.

5.2.3 Analyse von Einflussfaktoren auf den DRAM-Anteil

Eine Verringerung der Zugriffszeit des DRAM wirkt sich unmittelbar auf die Laufzeit einer Applikation aus. Eine weitere Betrachtung der Vergleichswerte erlaubt die nächste Feststellung:

Bei Verwendung des 3D-DRAM wird die relative Laufzeit für jede Applikation und jede Eingangsdatei auf etwa ein Viertel reduziert.

Der genau berechnete Faktor variiert zwar von Testfall zu Testfall, bleibt aber im überwiegenden Teil der Fälle in der Nähe die Zahl 4. Dieser Zusammenhang lässt sich unmittelbar durch das eingesetzte 3D-DRAM-Modell erklären. Das Modell versucht eine mögliche Leistungssteigerung durch Stapeltechnik zu schätzen. Aus datentechnischer Sicht ist es im Wesentlichen auf eine Reduktion der Zykluszeit zurückzuführen. In Abbildung 5.6 ist diese Annahme nochmal hervorgehoben.

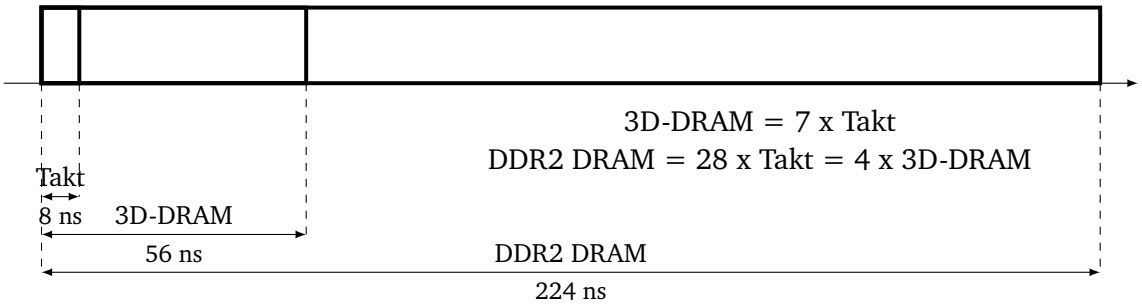


Abbildung 5.6: Vergleich der Zykluszeit von DDR2- und 3D-DRAM in Relation zum Taktabstand.

Die Reduktion der Zykluszeit bei einem Zugriff auf den DRAM wirkt sich somit in der gleichen Größenordnung auf die relative Verzögerung aus. Diese Erkenntnis überrascht wenig, wenn man bedenkt, dass der DRAM-Modellwechsel keine Änderung der notwendigen DRAM-Zugriffe bewirkt. Es kommt zu keiner Änderung, da die Größe des übermittelten Datenpakets konstant bleibt. Jede Verringerung der Zugriffszeit durch Stapeltechnik steigert somit die Leistungsfähigkeit des Systems.

Außerdem ist ein Blick auf eine weitere Größe interessant: die tatsächlichen Werte der relativen Verzögerung. Diese Werte sind vergleichsweise gering und bewegen sich für reale Applikationen im einstelligen Prozent- beziehungsweise Promillebereich. Selbst für die synthetische Funktion mit einer schlechten Datenlokalität beträgt die relative Verzögerung etwa 50 %. Im schlimmsten Fall würde ein DRAM ein ideales System also um die Hälfte der Laufzeit verzögern. Für reale Applikationen sind es 1 % bis 5 %. Anders ausgedrückt: Selbst wenn die Stapeltechnik in der Lage wäre, jede DRAM-Verzögerung zu vermeiden, würde das Gesamtsystem nur geringfügig schneller laufen.

Diese Feststellung lässt sich sehr gut anhand des Gesetzes von Amdahl [Amdahl, 1967] erklären. In der Abbildung 5.6 ist die Zykluszeit eines DRAM-Zugriffs im Verhältnis zur Taktbreite eingezeichnet. Ein Zugriff dauert zwar im Falle von DDR2 28 Takte, diese Zugriffe finden jedoch zu selten statt. Daher fällt ihr Gesamtanteil gering aus. Während des größten Teils der Ausführungszeit arbeitet die Applikation mit dem Pufferspeicher. Sie benötigt den DRAM in erster Linie am Anfang der Berechnung, wenn alle Instruktionen und Daten geladen werden, daraufhin mit langen Abständen während der Berechnung und zum Schluss, um die Daten wieder zurückzuschreiben. Eine hohe Verbesserung eines geringen Teils des Systems bringt nach Amdahl nur eine geringe Gesamtverbesserung.

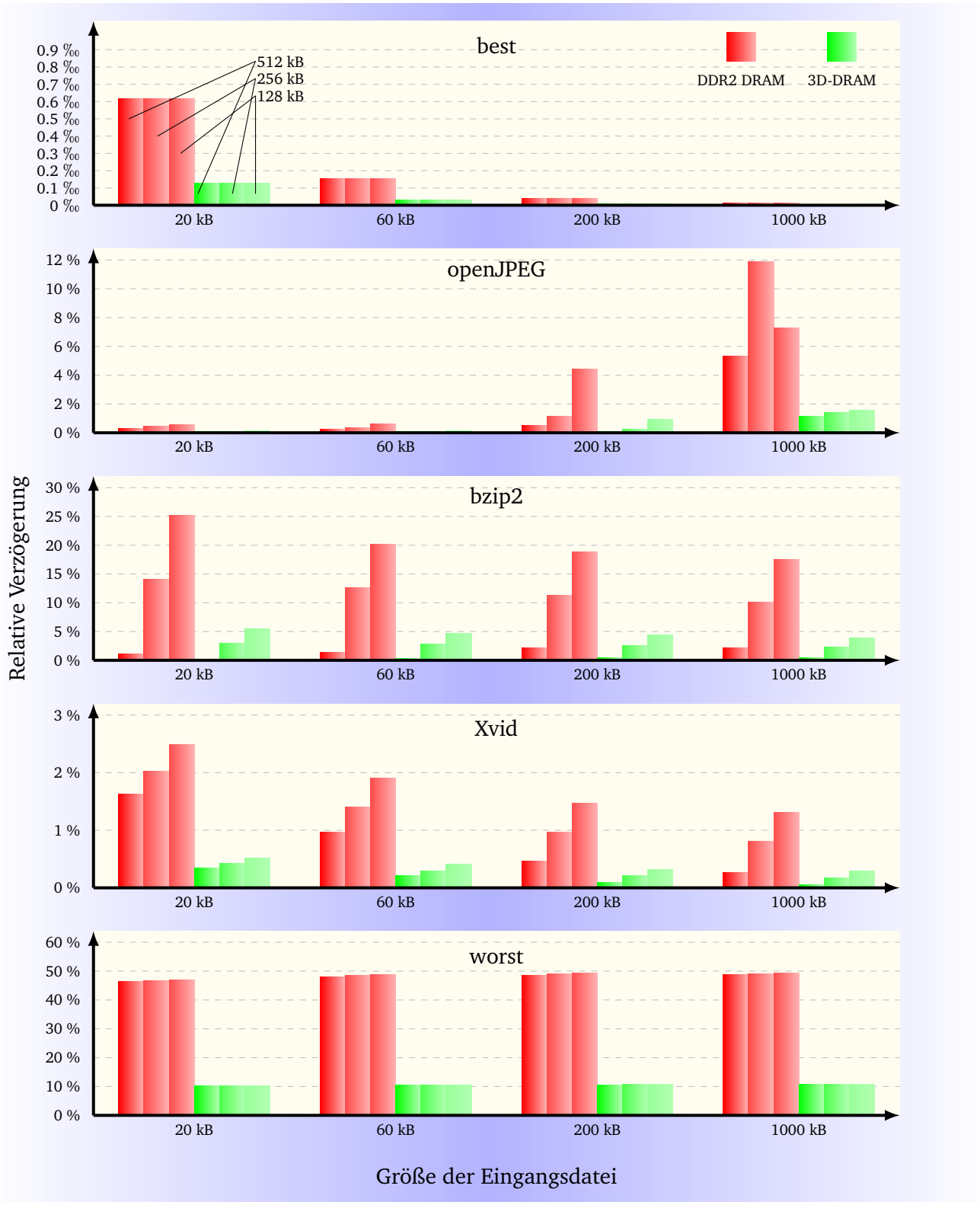


Abbildung 5.7: Vergleich der relativen Verzögerung zwischen einem System mit DDR2-DRAM und einem mit 3D-DRAM in Abhängigkeit von der Applikation, der Größe der Eingangsdatei und der Pufferspeicherkapazität

Die größte Auswirkung durch die Stapeltechnik kann also nur dann beobachtet werden, wenn der Gesamtanteil der DRAM-Nutzung größer wird. Dies kann dadurch erreicht werden, dass die Betriebsfrequenz zunimmt und ein Zugriff somit mehr Takte dauert. Die nächste Untersuchung dieser Arbeit umfasst daher die Erfassung von einer Steigerung des DRAM-Anteils durch höhere Taktfrequenz.

Durch den Vergleich mit DDR2 wird deutlich, dass die Größe der Eingangsdatei eine Rolle bei der Bestimmung des DRAM-Anteils spielt. Es wäre also von Interesse, die Untersuchung auch für größere Eingangsdaten durchzuführen. Leider bietet das Modell insbesondere mit Berücksichtigung der realen Simulationszeiten keinen Spielraum für eine weitere Erhöhung des Umfangs der Daten. Daher wurde für diese Arbeit eine indirekte Methode gewählt.

Eine Datei wird vor der Verarbeitung eingelesen und befindet sich dann im Pufferspeicher. Wenn die Kapazität des Pufferspeichers allerdings nicht ausreicht, um die Datei vollständig aufzunehmen, werden zusätzliche DRAM-Zugriffe initiiert. Je größer die Datei ist, desto mehr Daten müssen im Pufferspeicher ausgetauscht werden. Daher kann grob Folgendes angenommen werden:

Annahme 5

Eine Verringerung der Kapazität des Pufferspeichers um Faktor 2 emuliert in etwa eine Vergrößerung der Eingangsdaten um den gleichen Faktor.

Basierend auf dieser Annahme wurde der Vergleich der DRAM-Systeme mit der Pufferspeicherkapazität auf der zweiten Ebene in Höhe von 512 kB, 256 kB und 128 kB durchgeführt. Abbildung 5.7 zeigt die gemessenen Werte. Auf eine Kennzeichnung der maximalen und minimalen Werte wurde aus Einfachheitsgründen verzichtet.

Bei der ersten Applikation mit der guten Datenlokalität scheint die Reduktion der Pufferspeicherkapazität keine Wirkung zu zeigen. Wenn man aber die getesteten Dateigrößen betrachtet, dann sieht man, dass sie sogar mit reduzierter Kapazität vollständig vom Pufferspeicher aufgenommen werden können. Bei der schlechten Datenlokalität zeigt sich eine Unabhängigkeit von Größe der Eingangsdaten und zusätzlich von der Pufferspeicherkapazität. Sowohl bei einem großen als auch bei einem kleinen Pufferspeicher treten Schwierigkeiten im Umgang mit Zufallsadressen auf.

Für reale Applikationen gilt zwar der gleiche Zusammenhang zwischen der Größer der Eingangsdatei und der Pufferspeicherkapazität, sie reservieren aber in der Regel viel mehr Speicherplatz. Daher lässt sich für alle Applikationen folgende Entwicklung erkennen:

Eine Verringerung der Pufferspeicherkapazität führt zu einer Zunahme des DRAM-Anteils an der Gesamtausführung und somit zu einer Zunahme der relativen Verzögerung.

Insbesondere für das bzip2-Programm können Werte ermittelt werden, die in der Größenordnung der schlechten Datenlokalität liegen. Insgesamt kann unter Berücksichtigung der Annahme 5 Folgendes festgehalten werden:

Der DRAM-Anteil und damit die Möglichkeiten der Leistungssteigerung durch Stapeltechnik sind abhängig von der Größe der Eingangsdaten. Sie profitieren von größeren Datenmengen.

5.3 Erhöhung der Taktfrequenz

Neben der Erhöhung der Menge der Eingangsdaten sollte sich der DRAM-Anteil auch durch eine Erhöhung der Differenz zwischen der Taktbreite und der DRAM-Zykluszeit beeinflussen lassen. Wenn die Betriebsfrequenz zunimmt, benötigt ein (gleichbleibend schneller) DRAM-Zugriff mehr Takte. So lässt

sich der DRAM-Anteil theoretisch beliebig steigern. Eine steigende Betriebsfrequenz führt allerdings zur höherer Abwärme, die ab einem bestimmten Wert einen Aufwand für die Abführung verlangt, der aus wirtschaftlichen Gründen nicht mehr vertretbar ist [Rauber und Rünger, 2008]. Daher ergibt sich eine Obergrenze für die Leistungssteigerung des DRAM.

Die längste reale Simulationszeit benötigte das System mit dem DDR2-DRAM-Modell. Das 3D-DRAM-Modell konnte in erheblich kürzerer Zeit evaluiert werden. Daher wurden für diese Arbeit noch weitere Testfälle hinzugefügt. In einem Fall wurde die Pufferspeicherkapazität weiter reduziert: von 512 kB auf 32 kB. Der letzte Wert entspricht der Kapazität auf der ersten Ebene. Außerdem wurde die Laufzeit für zwei Betriebsfrequenzen gemessen: für 1000 MHz und für 4000 MHz. Der letzte Wert repräsentiert eine Obergrenze für reale Systeme. In den Abbildungen 5.8 und 5.9 werden die Werte absichtlich nebeneinander gestellt.

Die erste Beobachtung scheint für alle Applikationen zu gelten:

> Der DRAM-Anteil wächst mit der Größe der Eingangsdaten. Die Zunahme ist stark applikationsabhängig.

Das XviD-Programm zeigt eine stetige Zunahme des DRAM-Anteils. Die absoluten Werte werden allerdings mit zunehmender Größe kleiner. OpenJPEG zeigt den schon zuvor beobachteten Sprung bei einer bestimmten Größe. Bei 60 kB Bilddaten ist der Sprung bei 32 kB Pufferspeicherkapazität zu beobachten. Er verschiebt sich im Bild nach rechts. Die größte Zunahme des DRAM-Anteils verzeichnet das bzip2-Programm. Diese Applikation ist auch schon in vorhergehenden Untersuchungen durch besonders hohe Werte aufgefallen.

Eine weitere Auffälligkeit ist folgende:

> Die Zunahme des DRAM-Anteils ist proportional zur Zunahme der Betriebsfrequenz.

Dieses Verhalten ist darauf zurückzuführen, dass der Prozessor während eines DRAM-Zugriffs angehalten wird, sodass die Abstände zwischen den Zugriffen nur durch die Taktbreite beeinflusst werden und entsprechend proportional schrumpfen. An dieser Stelle sei angemerkt, dass der Pufferspeicher bei den Untersuchungen nicht verändert wurde. Seine Implementierung ist zyklusgenau, sodass er keinen direkten Einfluss auf die Ausführungszeit durch eine Veränderung der Taktrate ausübt. Für reale Systeme kann dieser Ansatz nicht angewendet werden, da der Pufferspeicher für höhere Frequenzen angepasst werden muss. Für diese Arbeit wird sein Anteil herausgerechnet. Die Architektur des Pufferspeichers hat dennoch einen Einfluss auf die Kommunikation mit dem DRAM. Daher sind die Werte als Abschätzungen zu betrachten.

Wenn die Stapeltechnik eine Verbesserung des DRAM erreichen soll, wird diese Verbesserung am deutlichsten bei größter Betriebsfrequenz und hohen Eingangsdaten vorliegen. Je kleiner diese Parameter sind, desto geringer ist der Gewinn durch die neue Technologie.

5.4 Evaluation des lokalitätsbasierten Ansatzes

5.4.1 Die Idee hinter dem Ansatz

Der Vergleich des 3D-DRAM-Modells mit einem DDR2-DRAM-Modell zeigte, dass sich eine Reduktion der Zykluszeit direkt auf die Gesamtlaufzeit der Ausführung auswirkt. Bei gleichbleibender Datenbreite ist die Wirkung proportional. Ausgehend von dieser Beobachtung scheint der nächste Schritt darin zu liegen, die DRAM-Zugriffszeit weiter zu reduzieren. Der Aufbau der DRAM-Bitfelder setzt hier allerdings Grenzen (Abschnitt 2.1.3). Darum kann dieser Ansatz nicht auf den gesamten DRAM angewendet werden. Bei der Beschreibung des 3D-DRAM-Modells im Abschnitt 3.3.3 wurde bereits eine weitere

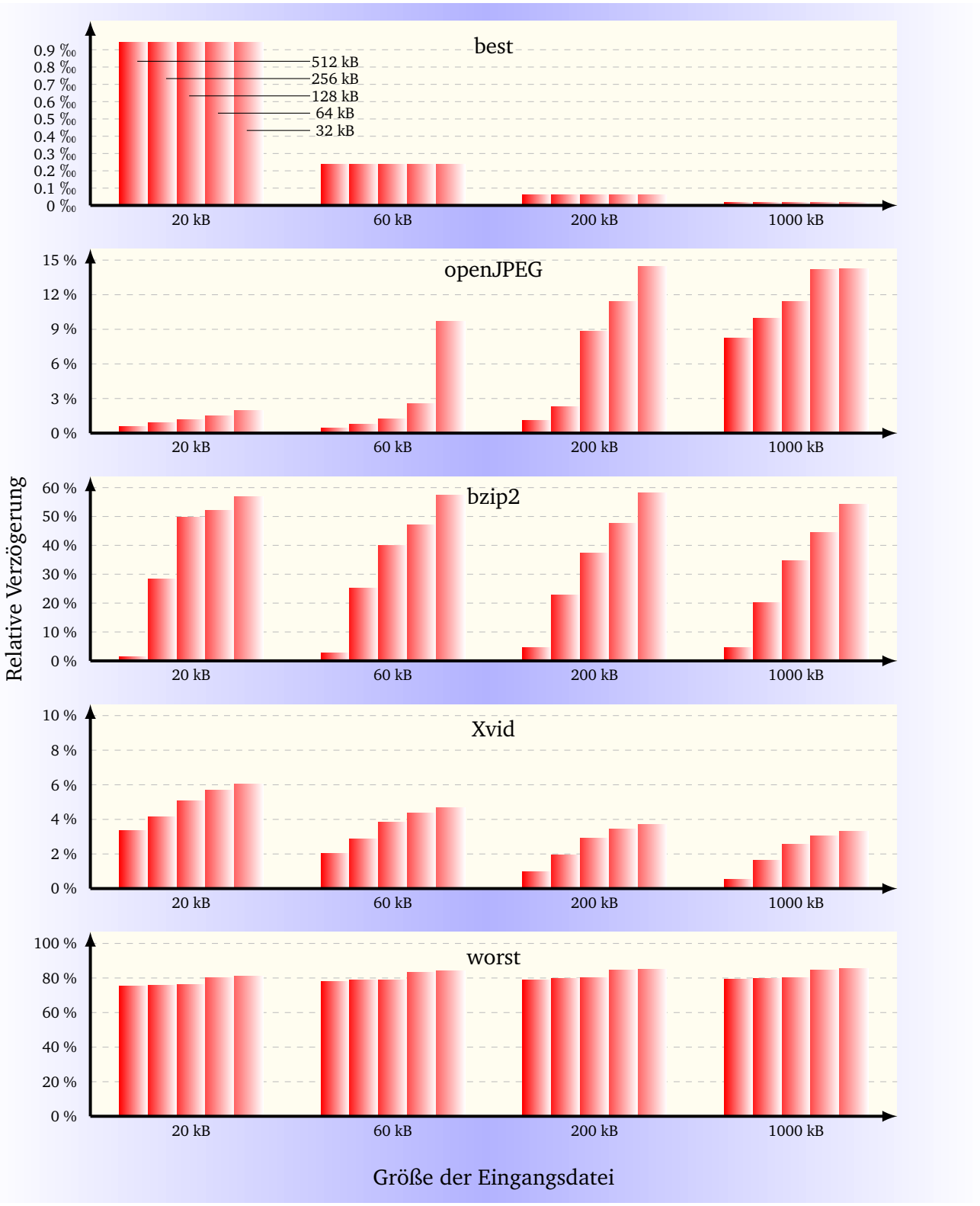


Abbildung 5.8: Relative Verzögerung durch DRAM in Abhängigkeit von der Applikation, der Größe der Eingangsdatei und der Pufferspeicherkapazität. Betriebsfrequenz ist 1 GHz

Abbildung 5.9: Relative Verzögerung durch DRAM in Abhängigkeit von der Applikation, der Größe der Eingangsdatei und der Pufferspeicherkapazität. Betriebsfrequenz ist 4 GHz

Möglichkeit vorgestellt: eine zusätzliche latenzoptimierte Schicht. Diese spezielle Schicht unterscheidet sich von den anderen durch eine halbierte Zykluszeit und eine daraus resultierende geringere Kapazität. In der Modellierung wird diese Reduktion auf ein Viertel geschätzt. Bei dieser Größenordnung sollte die anvisierte Zykluszeit realisierbar sein. Die übrig gebliebene Kapazität ist dennoch groß genug, um beispielsweise den kompletten Instruktionskode aufnehmen zu können.

Die erste Frage, die sich im Zusammenhang mit der latenzoptimierten Schicht stellt, ist folgende: Wie wirkt sich diese Schicht auf die Ausführungszeit aus? Die zusätzliche Reduktion der Zykluszeit sollte auf den ersten Blick messbare Ergebnisse produzieren. Das Lokalitätsprinzip zeigt allerdings, dass diese Reduktion nur beim ersten Zugriff auf die Daten eine Rolle spielt. Die häufig genutzten Strukturen bleiben im Pufferspeicher. Das System benötigt in diesem Fall keinen schnelleren DRAM. Die Ergebnisse der Untersuchungen, die in vorhergehenden Abschnitten präsentiert wurden, zeigen allerdings, dass der DRAM-Anteil durch eine steigende Betriebsfrequenz und durch eine steigende Größe der Eingangsdatei zunehmen kann. Wenn es zudem möglich ist, die Zugriffe, die für diesen Anteil verantwortlich sind, in die latenzoptimierte Schicht zu verlagern, sollte sich die geringe Zykluszeit unmittelbar bemerkbar machen.

Der erfolgreiche Ansatz, den Pufferspeicher in mehrere Ebenen einzuteilen, zeigt, dass das Lokalitätsprinzip für alle diese Ebenen greift, auch wenn die Daten und Häufigkeiten unterschiedlich sind. Diese Überlegung führt zu folgender Annahme

Annahme 6

Das Lokalitätsprinzip zeigt sich auf jeder Ebene des Speichersystems, wobei für jede Ebene bestimmte Daten und Zugriffshäufigkeiten ermittelt werden können.

Die Aufgabe liegt also darin, diese Daten zu ermitteln. Beim Pufferspeicher geschieht das automatisch, indem häufig genutzte Daten nicht ersetzt werden. Eine Möglichkeit, diesen Ansatz auf den DRAM auszuweiten, präsentiert [Lee et al., 2015]. In dieser Arbeit wird eine andere Möglichkeit evaluiert. Der automatische Ansatz passt sich zwar an jedes System an, erfordert aber eine zusätzliche Logik, die direkt im kritischen Pfad platziert werden muss und so der Zykluszeitreduktion gerade entgegen wirkt. Auf diese Logik kann verzichtet werden, wenn die Daten bereits per Software an die „richtige“ Stelle geschrieben werden. Die Architektur des 3D-DRAM sieht nur eine Aufteilung des Adressraums vor. Jede Schicht, auch die latenzoptimierte, agiert aber für sich als üblicher DRAM-Speicher.

Der Ansatz erfordert also eine Analyse der Applikation vor der Ausführung, um die häufig genutzten Datenstrukturen zu erkennen. Diese Daten werden dann entweder dynamisch innerhalb eines zweiten *heap* oder statisch per Linker in die latenzoptimierte Schicht platziert. Je größer die Eingangsdaten sind, und je höher die Betriebsfrequenz ist, desto größer sollte der Leistungsgewinn ausfallen.

5.4.2 Detektion der häufig genutzten Datenstrukturen

Während der Ausführung einer Applikation werden die Daten in der Regel zu Datenstrukturen zusammengefasst. Diese Strukturen können dann neben den eigentlichen Verarbeitungsdaten noch weitere Statusinformationen, Adressen und Verarbeitungsparameter enthalten. Das Lokalitätsprinzip postuliert nur, dass bestimmte Daten häufig gebraucht werden. Wenn man diese Daten analysiert, lassen sich eventuell auch häufig genutzte Datenstrukturen erkennen. Wie bei der Bestimmung der Lokalität können sowohl Schätz- als auch Erfassungsmethoden zum Einsatz kommen. Die Erfassungsmethoden stehen dabei in dem Ruf, sehr zeit- und speicherintensiv zu sein. Es stellt sich allerdings die Frage, wie viele Daten gesammelt werden müssen, um alle notwendigen Strukturen erkennen zu können.

Doch bevor diese Frage angegangen wird, soll zunächst die Methode für die Entdeckung dieser Strukturen beschrieben werden. Alle Testapplikationen reservierten Speicherplatz auf dem *heap* mittels vorde-

finierter C-Standardfunktionen. Sobald diese Funktionen allerdings aufgerufen wurden, war eine direkte Rückverfolgung auf die jeweilige Datenstruktur nicht mehr möglich. Als Rückgabewert wurde ein Zeiger erzeugt, der dann intern der jeweiligen Datenstruktur zugeordnet werden konnten. Wenn man also die Speichernutzungsdaten betrachtet, lassen sich Adressen im *heap* feststellen, die besonders häufig angesprochen wurden. Welchen C-Strukturen diese Adressen allerdings entsprechen, ist nicht mehr erkennbar. An dieser Stelle ist eine zusätzliche Softwarelösung erforderlich, die die Zuordnung auflöst. Auf eine tiefer gehende Suche in diesem Bereich wurde in dieser Arbeit allerdings verzichtet. Die Datenstrukturen wurden jeweils einzeln für die Applikationen ermittelt. Dazu wurde die VHDL-Testumgebung um ein Modul erweitert, das alle Änderungen des *heap*-Zeigers protokolliert. Die häufig genutzten Datenadressen konnten somit einem entsprechenden Wert dieses Zeigers zugeordnet werden. Jede Änderung wurde mit einem Zeitstempel aufgenommen, sodass der Zeitpunkt während der Ausführung feststellbar war. Dieser Zeitpunkt führte zu der jeweiligen Applikationsfunktion, die die `malloc`-Funktion benutzte. Ein Blick in den Quellkode erlaubte dann die nachträgliche Zuordnung von der Speicheradresse zu der Datenstruktur. Da die Datenstrukturen während der Ausführung freigegeben werden, konnten dem gleichen Wert des *heap*-Zeigers mehrere Daten zugeordnet werden. Daher wurde speziell für diese Untersuchung die Freigabefunktion abgeschaltet.

Die Erfassung der Speichernutzungsdaten erfolgte mittels der eigenen Softwarelösung *3DMemory* [Schoenberger und Hofmann, 2014a]. Als Referenzwert für die Analyse diente die Verarbeitung einer der Eingangsdaten in der jeweils kleinsten Größe. Die dabei entdeckten Datenstrukturen und ihr Anteil an der Anzahl aller Datenzugriffe stellten den Ausgangspunkt für die Analyse dar. Im Verlauf der Analyse wurde die Größe der Eingangsdaten verringert und die Menge der erkannten Datenstrukturen erfasst. Abbildung 5.10 zeigt die Ergebnisse der Analyse.

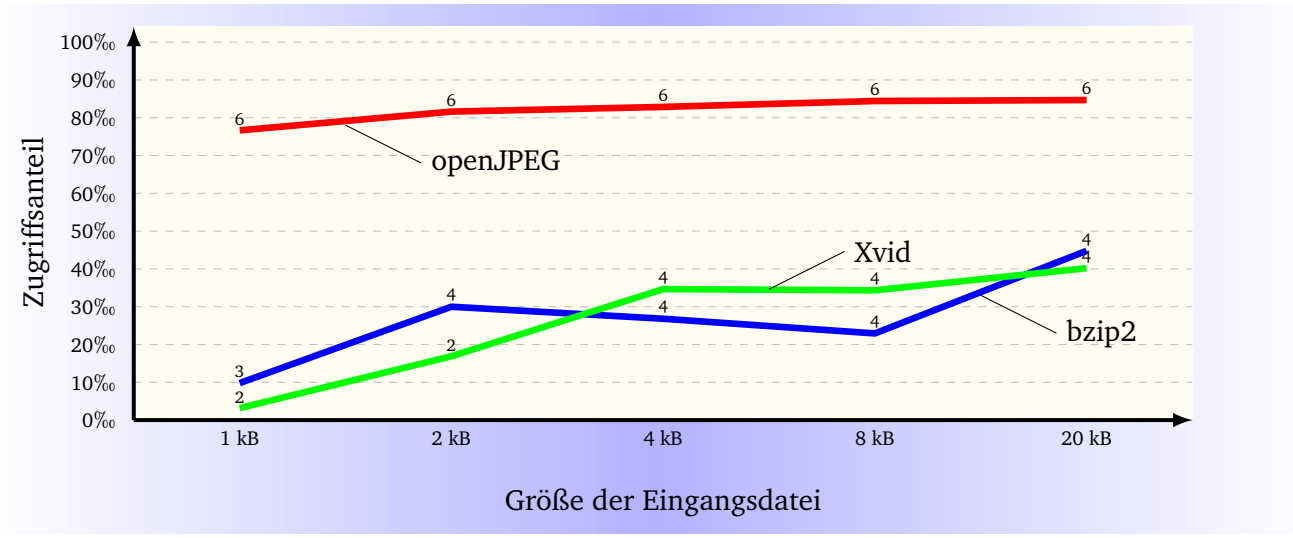


Abbildung 5.10: Anzahl der erkannten, häufig genutzten Datenstrukturen (die Zahl oberhalb der Messpunkte) und ihr Anteil in Bezug auf alle Datenzugriffe (die Linien) in Abhängigkeit von der Größe der Eingangsdatei [Schoenberger und Hofmann, 2014b]

Zu jeder Größe der Eingangsdatei sind zwei Werte zugeordnet: eine Zahl, die für die Anzahl der erkannten Datenstrukturen steht und die Summe aller Zugriffe auf diese Strukturen in Bezug auf alle Zugriffe auf den *heap*. OpenJPEG zeigt die besten Ergebnisse. Selbst für die kleinste Datei (entspricht einem Bild mit 16x16 Bildpunkten) lassen sich bereits alle Datenstrukturen erkennen. Ihr Anteil liegt bei fast 80 % aller Zugriffe. Bei bzip2 sind mindestens 2 kB an Daten erforderlich und bei XviD 4 kB. Es lassen sich zwar noch weitere häufig genutzte Daten feststellen, ihr Anteil ist aber vergleichsweise klein. Im Anhang A.2 sind die entdeckten Datenstrukturen und die jeweiligen Funktionen der Applikationen

aufgelistet. Der Abbildung lässt sich allerdings noch eine weitere Entwicklung entnehmen: Der relative Anteil aller Zugriffe scheint mit der Zunahme der Eingangsgröße für bzip2 und XviD zu steigen. Eine Platzierung dieser Daten in der latenzoptimierten Schicht könnte also für größere Eingangsdaten eine größere Leistungssteigerung erreichen.

Insgesamt zeigt die Detektion der häufig genutzten Datenstrukturen, dass diese bereits für kleine Datenmengen „sichtbar“ werden. Außerdem bringen Analysen größerer Datenmengen keine neuen Erkenntnisse.

5.4.3 Reduktion der Verzögerung

Die latenzoptimierte Schicht weist zwar eine geringere Kapazität als andere Schichten des gestapelten 3D-DRAM auf, ihre Größe lässt es jedoch zu, für alle Testapplikationen den gesamten Instruktionscode innerhalb dieser Schicht zu platzieren. Weiterhin ist es möglich den *heap* in zwei Teile aufzuspalten und die häufig genutzten Datenstrukturen ebenfalls in der latenzoptimierten Schicht zu verorten. Dazu sind allerdings spezielle Reservierungs- und Freigabefunktionen erforderlich. Im vorhergehenden Abschnitt wurde die Detektion dieser Datenstrukturen erläutert. Ihre Anzahl ist mit 4 bis 6 pro Applikation relativ gering, sodass der Anpassungsaufwand überschaubar bleibt. Für die Speicherverwaltung wurden drei zusätzliche Funktionen implementiert:

5.4.3 Speicherverwaltung des *heap* in der latenzoptimierten Schicht

```
void * fast_malloc(                size_t size);
void   fast_free(    void * ptr);
```

Die Reservierung sowie die Freigabe der häufig genutzten Datenstrukturen wurden an den jeweiligen Stellen im Quellkode angepasst. Die Übersetzung des Kodes durch den Compiler führte allerdings nicht dazu, dass der Funktionsaufruf an der jeweiligen Stelle durch eine andere Adresse ersetzt wurde. Stattdessen kam es zu teilweise starken Änderungen des Maschinenkodes. Die Ursache für dieses Verhalten ist im Compileraufbau zu suchen und wurde nicht weiterverfolgt. Für die Analyse bedeutete dies allerdings, dass die zuvor ermittelten Zykluszahlen und der Pufferspeicheranteil nicht verwendet werden konnten. Daher mussten diese Werte für die Analyse nochmals erfasst werden. Die genauen Zahlen sind im Anhang A.3 aufgelistet. In Abbildung 5.11 sind die jeweiligen Werte für die relative Verzögerung dargestellt.

In der Abbildung werden drei Messwerte miteinander verglichen:

- (links) System mit 3D-DRAM
- (mitte) System mit 3D-DRAM und latenzoptimierter Schicht
- (rechts) System mit lokalitätsbasiertem Ansatz

Des Weiteren sind die Werte für unterschiedliche Pufferspeicherkapazitäten und für unterschiedliche Größen der Eingangsdaten dargestellt. Die latenzoptimierte Schicht wird als FMR (***F**ast **M**emory **R**egion*) bezeichnet. Die Betriebsfrequenz entspricht 4 GHz. Die Daten repräsentieren somit die obere Grenze für die Leistungssteigerung durch den vorgestellten Ansatz.

Es lassen sich große Unterschiede zwischen den Applikationen feststellen. Die Werte für 3D-DRAM wurden bereits in vorherigen Untersuchungen präsentiert. Darum werden sie im Weiteren nicht mehr näher erläutert. Zunächst soll ein System fokussiert werden, dass nur die latenzoptimierten Schicht aufweist, ohne Softwareeinsatz. Bei dieser Konstellation beinhaltet die Schicht nur die Instruktionsdaten, die Konstanten und globale Variablen. Für openJPEG und bzip2 lassen sich keine Unterschiede durch den Einsatz der Schicht erkennen. Die relative Verzögerung bleibt für alle Änderungen der Pufferspeicherkapazität und für jede Größe der Eingangsdaten gleich. Daraus lässt sich schlussfolgern, dass Daten

der latenzoptimierten Schicht entweder nicht so häufig gebraucht werden oder bereits nach einmaligem Einlesen innerhalb des Pufferspeichersystems bleiben, sodass keine DRAM-Zugriffe erforderlich sind. Die reduzierte Zykluszeit der Schicht spielt für diese Daten keine Rolle.

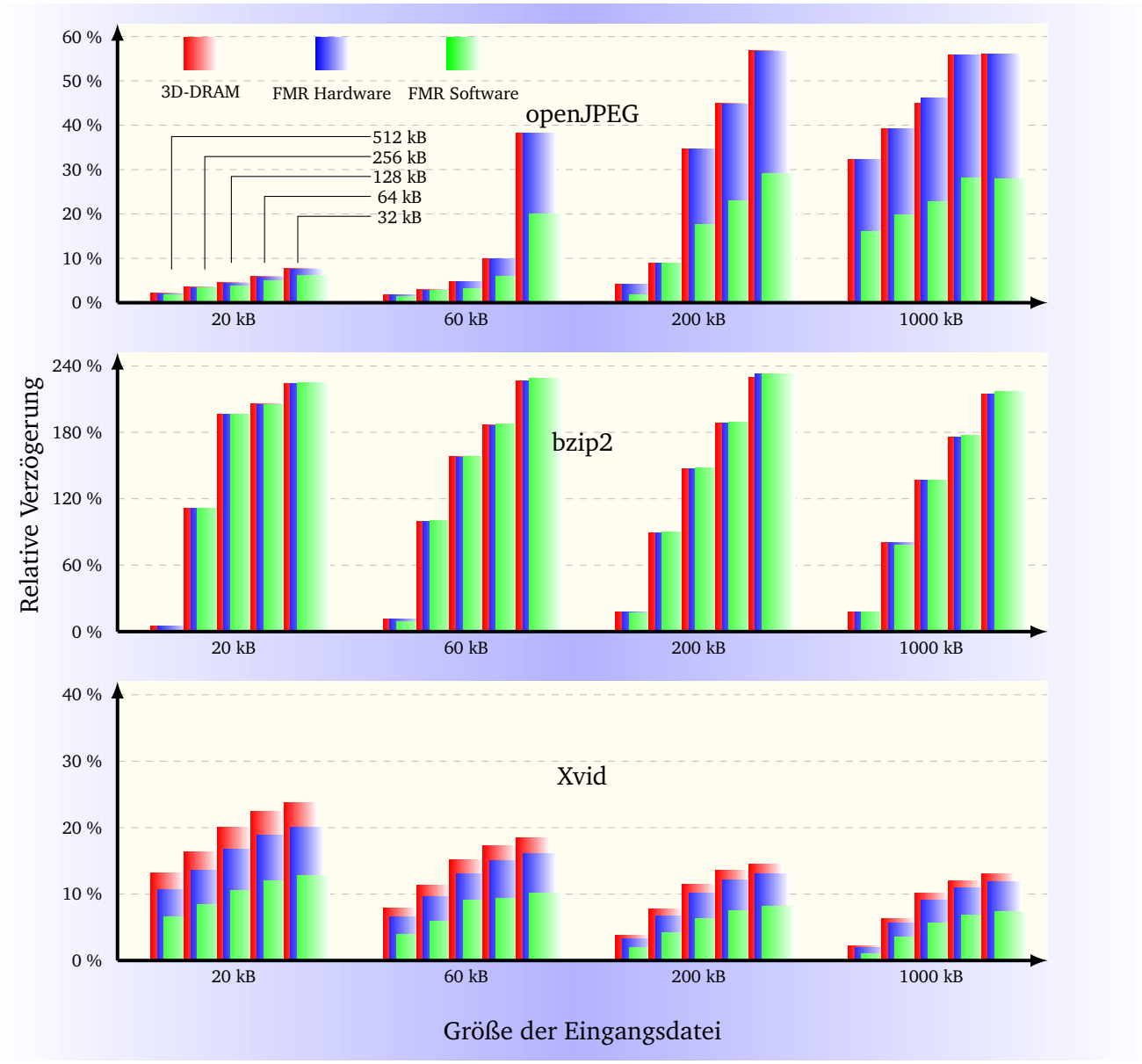


Abbildung 5.11: Einfluss der latenzoptimierten Schicht (FMR), mit und ohne den lokalitätsbasierten Ansatz, auf die relative Verzögerung durch den DRAM [Schoenberger und Hofmann, 2015]

Ein anderes Bild ergibt sich bei XviD. Die relative Verzögerung lässt bereits durch die Integration der latenzoptimierten Schicht reduzieren. Der Anteil und der Reduktionseffekt sinken zwar für größere Eingangsdaten, bleiben aber nahezu in allen Testfällen messbar. Dies lässt sich damit erklären, dass die Applikation auf eine große Menge an Verarbeitungskonstanten zurückgreift, die in der latenzoptimierten Schicht zusammen mit den Instruktionsdaten zu finden sind. Die Reduktion der relativen Verzögerung ist in der gleichen Weise abhängig von der Pufferspeicherkapazität und der Größe der Eingangsdaten wie bei einem nicht modifiziertem 3D-DRAM.

Die Werte für den lokalitätsbasierten Ansatz - wenn die häufig genutzten Datenstrukturen mittels spezieller Befehle in der latenzoptimierten Schicht platziert wurden - zeigen ebenfalls applikationsab-

hängige Ergebnisse. Das bzip2-Programm erweist sich zum wiederholten Mal als ein Sonderfall. Die Platzierung der Datenstrukturen führt zu keinerlei Reduktion der relativen Verzögerung. Die Ergebnisse sind sogar geringfügig schlechter als ohne den Ansatz. Wie bereits in vorhergehenden Untersuchungen vermutet wurde, scheint diese Applikation einem zufälligen Zugriff auf die Daten am nächsten zu kommen, sodass die Datenlokalität sehr gering ist. Selbst die reduzierte Zykluszeit der latenzoptimierten Schicht vermag hier keine messbare Verbesserung zu erreichen. Ganz anders verhält es sich dagegen bei openJPEG und XviD. Für beide Applikation ist eine sichtbare Abnahme der relativen Verzögerung feststellbar. Die Abhängigkeit von der Pufferspeicherkapazität und damit von der Größe der Eingangsdaten ist gleichbleibend wie bei einem System mit nur 3D-DRAM. Die Werte für die relative Verzögerung sind aber in etwa halbiert.

6 Speichernutzung auf einem Mehrkernsystem

6.1 Parallelisierungsgüte

Die treibende Kraft für den Fortschritt von integrierten Schaltungen war lange Zeit das mit der Miniaturisierung verbundene Wachstum der Betriebsfrequenz. Die Verkleinerung der physikalischen Abmessungen von digitalen Komponenten erlaubte kürzere Durchlaufzeiten. Diese Entwicklung erreichte allerdings Mitte des ersten Jahrzehnts des 21. Jahrhunderts einen Zustand, der einen Paradigmenwechsel auslöste. Die Umschaltung der Transistoren verursachte bereits zuvor Wärme, die abgeführt werden musste. In dieser Zeit erreichte sie allerdings eine Höhe, deren Handhabung technologisch und wirtschaftlich nicht mehr vertretbar war [Rauber und Rünger, 2008]. Diese Grenze betraf aber nicht die weiter voranschreitende Miniaturisierung. Eine Lösung boten Mehrkernprozessoren an. Ihre Betriebsfrequenz war zwar begrenzt, ihre Anzahl konnte jedoch ohne großen Aufwand erhöht werden.

Während die Hardwareumsetzung der Mehrkernprozessoren vergleichsweise wenig Aufwand erforderte, ist es bei der entsprechenden Software anders. Dabei bietet die Parallelisierung von Applikationen ein enormen Leistungssteigerungspotenzial [Amdahl, 1967]. Das größte Problem bleibt nach wie vor, dass automatisierte Lösungen weit unter dem Optimum bleiben [Karkowski und Corporaal, 1997]. Die beste Parallelisierung lässt sich weiterhin durch einen Menschen erreichen, der durch Werkzeuge unterstützt wird. Es stellt sich zunächst die Frage, inwiefern dieser Exkurs einen Bezug zu dieser Arbeit hat, die sich ja hauptsächlich mit Speichern beschäftigt. Angesichts der aktuellen Entwicklungen kann eine Untersuchung auf einem Mehrkernsystem nicht ausgelassen werden. Im **Unterschied zum Einkernsystem** geht es hier allerdings primär um einen möglichen **alternativen Analyseansatz**, der zu einer Leistungssteigerung durch 3D-DRAM führen könnte.

Für die Untersuchungen werden allerdings eine parallelisierte Software und ein Mehrkernsystem benötigt. Die Parallelisierung von Software deckt ein breites Forschungsgebiet ab und kann nur ansatzweise betrachtet werden. Zudem werden die Ergebnisse nur für Testfälle benötigt. Daher wurde auf möglichst einfache und schnell implementierbare Lösungen Wert gelegt. Die Parallelisierungsgüte sollte dennoch einen vertretbare Höhe aufweisen, sodass zunächst der Gewinn durch die Parallelisierung betrachtet wird. Die Parallelisierungsansätze wurden bereits in Abschnitt 4.4.2 erläutert. In Abbildung 6.1 auf der nächsten Seite sind die Simulationsergebnisse für ideale Zwei- und Dreikernsysteme abgebildet.

Die Untersuchung ist auf **drei Kerne beschränkt**. Zum einen ist dies der Tatsache geschuldet, dass die Verarbeitung von Farbkoordinaten bei openJPEG parallelisiert wurde, sodass eine Erhöhung der Kernanzahl keine Auswirkungen auf die Verringerung der Laufzeit gehabt hätte. Zum anderen musste die reale Simulationszeit, insbesondere bei einem kompletten Speichersystem, berücksichtigt werden. Hinzu kam noch eine weitere Randbedingung hinzu. Die Reduktion der Laufzeit durch Parallelisierung ist zwar für alle Testapplikationen messbar, der Effekt ist allerdings jeweils unterschiedlich. Während openJPEG auf einem Dreikernsystem mit 40 % der Laufzeit, bezogen auf ein Einkernsystem, nahezu optimale Leistungssteigerung erreicht, ist bei XviD für feste Bildgröße mit 80 % der Laufzeit eine viel geringere Parallelisierungsgüte festzustellen. Es lässt sich zudem kein Unterschied zwischen Zweikern- und Dreikernsystemen messen. Durch die Variation der Videolänge ist der Parallelisierungsgewinn zwar größer, erreicht aber nicht die Werte von openJPEG oder bzip2. Da größere Eingangsdateien nicht in vertretbarer Zeit simuliert werden konnten, bleibt die Frage offen, ob die Parallelisierung für übliche Videosequenzen höhere Beschleunigung erreichen kann. Die gemessenen Resultate konnten jedenfalls nicht überzeugen, sodass Xvid für weitere Untersuchungen zunächst nicht berücksichtigt wurde.

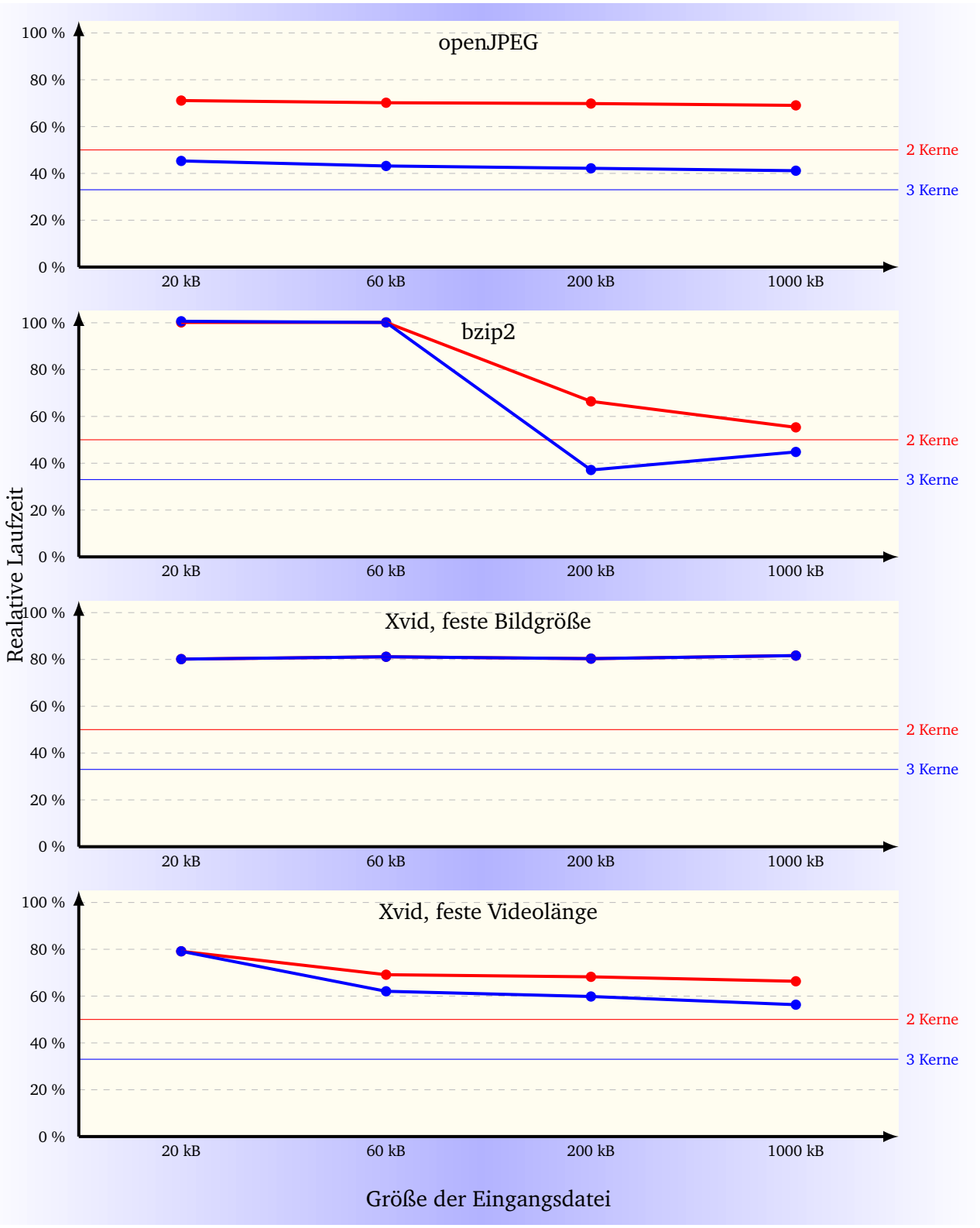


Abbildung 6.1: Relative Laufzeit der Testapplikationen, bezogen ein auf Einkernsystem. Maximale Werte für Zwei- und Dreikernsysteme sind zusätzlich eingezeichnet. Bzip2 und XviD aus [Dahlem, 2015].

Für die weiteren Untersuchungen wurden somit nur openJPEG und bzip2 verwendet. Das bzip2-Programm ist zwar erst ab 100 kB Eingangsdaten dazu in der Lage, diese auch parallel zu bearbeiten, die Reduktion der Laufzeit ist jedoch ähnlich wie bei openJPEG nahezu ideal. Eine Erhöhung der Kernanzahl hätte vermutlich bei bzip2 zu einer weiteren Beschleunigung geführt. Das Programm wäre aber als einziger Testfall übrig geblieben. Insgesamt lässt sich feststellen, dass die Parallelisierungsansätze selbst mit geringem Aufwand messbare und im Falle von openJPEG und bzip2 sehr gute Ergebnisse hervorbrachten.

6.2 Analyse der Speichernutzung

Die Hypothesen 2 und 3 postulieren, dass nur eine Ausnutzung des Lokalitätsprinzips zusätzliche Leistungssteigerung bringen kann. Insbesondere zur Überprüfung der dritten Hypothese ist ein alternativer Analyseansatz erforderlich. Abbildung 6.2 zeigt eine Analyse der Speichernutzung der Komprimierungs- und Dekomprimierungsroutinen von openJPEG, das die besten Parallelisierungsergebnisse zeigte.

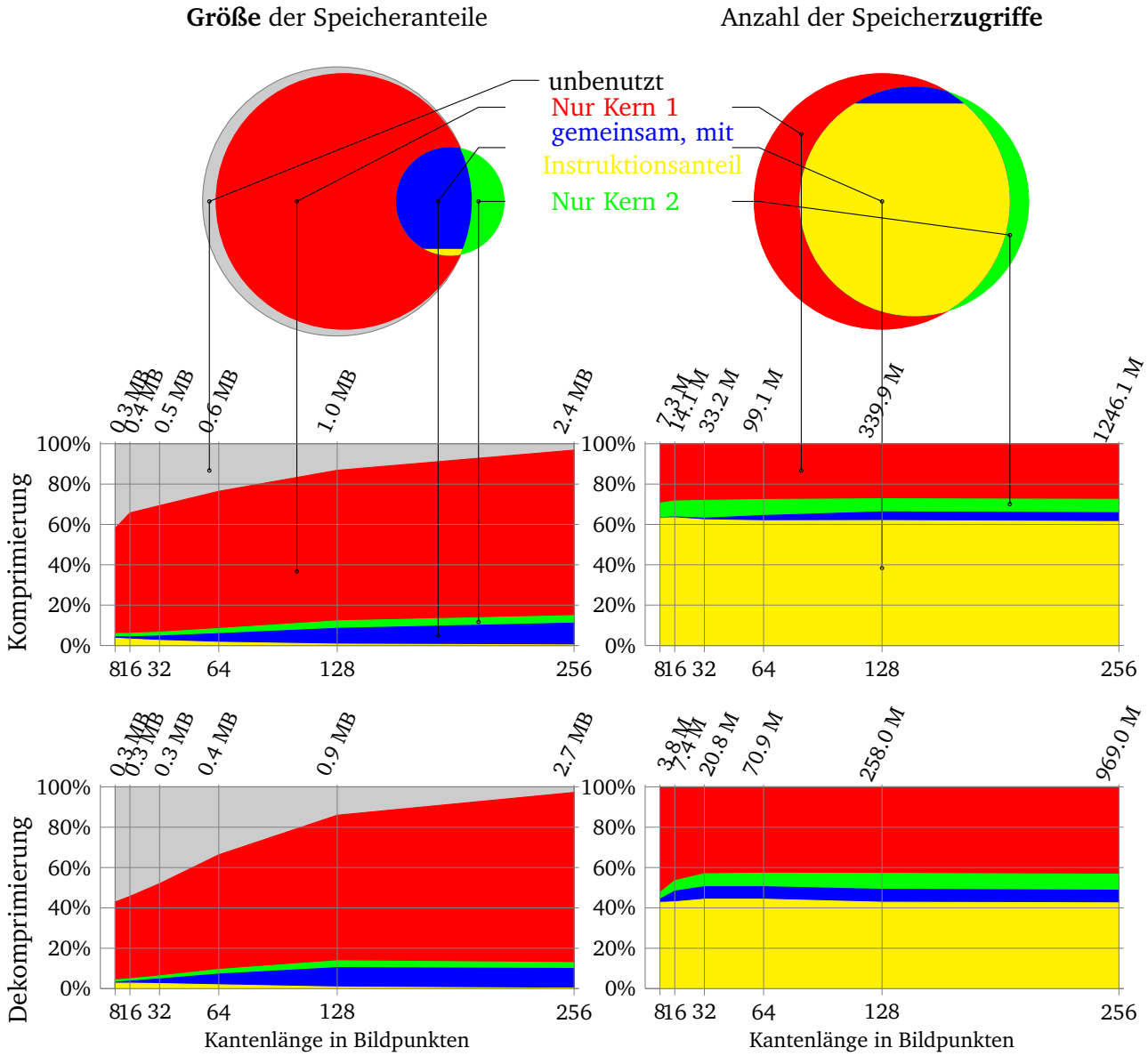


Abbildung 6.2: Größe der Speicheranteile (links) und die Anzahl der entsprechenden Zugriffe (rechts) für Kompressions- und Dekompressionsvorgänge von openJPEG auf einem Zweikernsystem

Die Abbildung zeigt auf der linken Seite die Verteilung des Speichers in Abhängigkeit von dessen Nutzung. Neben der Entwicklung der Verteilung - bezogen auf wachsende Kantenlänge der verarbeiteten Bilder - sind die Verhältnisse nochmals für die größten Eingangsdaten als Flächen der Kreise dargestellt. Manche Teile des openJPEG-Programms benötigen zwar Speicherplatz, werden aber während der Ausführung nicht gebraucht. Das sind für den Kompressionsvorgang die Funktionen der Dekomprimierung und jeweils umgekehrt. Dieser Anteil ist für kleine Eingangsdaten erwartungsgemäß groß und für größere Daten stark abnehmend. Für 256 Bildpunkte Kantenlänge beträgt dieser Anteil bei 2,4 MB beziehungsweise 2,7 MB des verwendeten Speichers weniger als 1 %.

Der größte Speicheranteil wird vom Hauptkern beansprucht, der die sequentiellen Programmteile abarbeitet. Diese Verhältnisse sind in erster Linie darauf zurückzuführen, dass der Hauptkern die Datenstrukturen für die Eingangsdaten reserviert. Der zweite Kern benötigt nur die aktuellen Verarbeitungsdaten. Diese sie sind im Vergleich zum Anteil des ersten Kerns viel kleiner. An dieser Stelle lässt sich die Lokalität der Daten bereits deutlich erkennen. Der zweite Kern wird nur bei parallelisierten Funktionen benötigt, die die meiste Ausführungszeit beanspruchen. Daher wird der zweitgrößte Speicheranteil von beiden Kernen benutzt. Dieser Anteil ist nochmals in Daten und Instruktionen unterteilt.

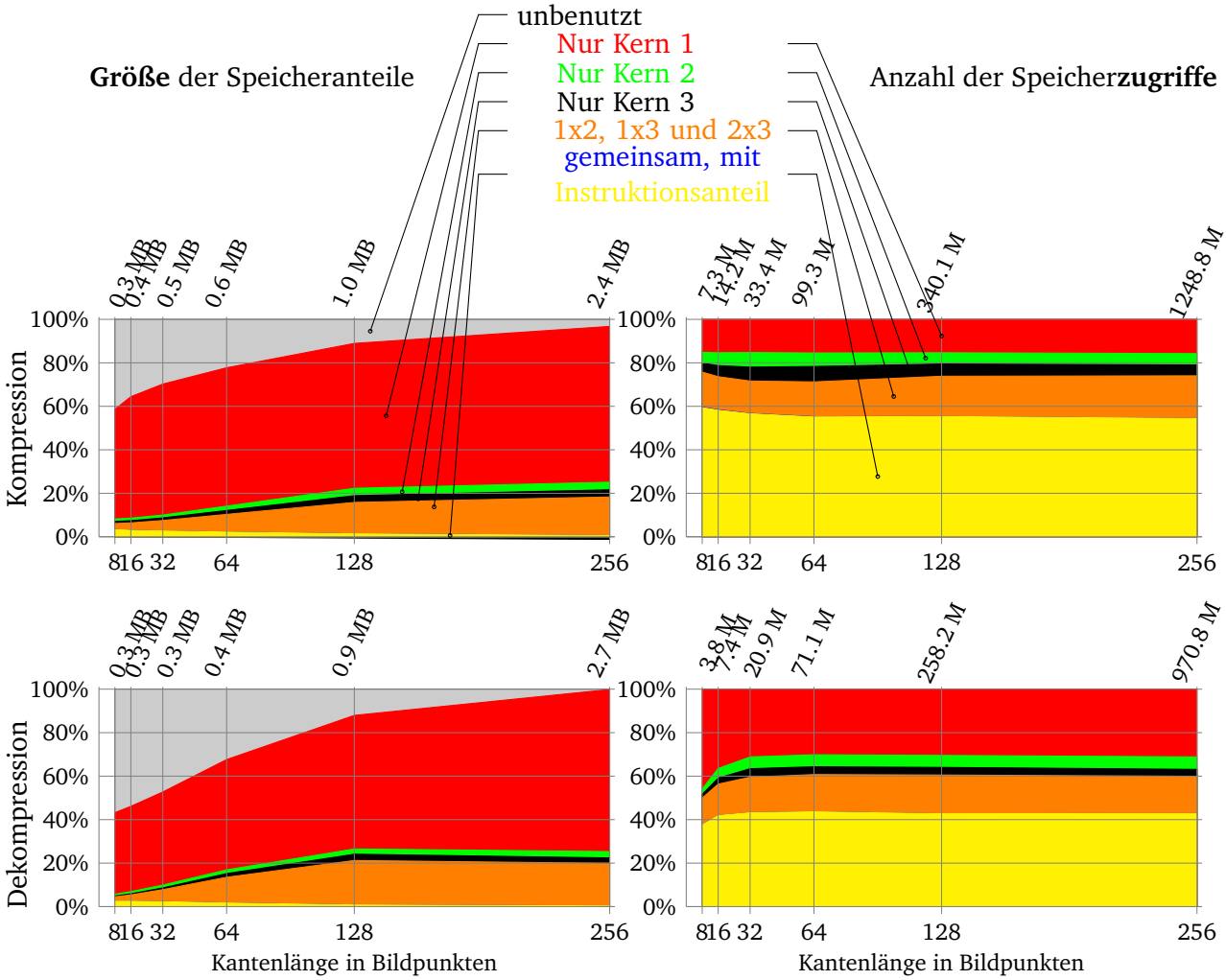


Abbildung 6.3: Größe der Speicheranteile (links) und die Anzahl der entsprechenden Zugriffe (rechts) für Kompressions- und Dekompressionsvorgänge von openJPEG auf einem Dreikernsystem

Die Absicht dieser Unterteilung wird auf der rechten Seite der Abbildung deutlich. Während die von beiden Kernen gemeinsam genutzten Instruktionen und damit der Speicher weniger als 1 % der verwendeten Kapazität ausmachen, dominiert dieser Anteil bei Zugriffen auf der rechten Seite. Insbesondere

bei der Kompressionsroutine liegen über 60 % aller Zugriffe im gemeinsam genutzten Instruktionsspeicher. Der nur vom Hauptkern genutzte Speicher wird zwar auch in nicht zu vernachlässigbarem Umfang angesprochen. Das Verhältnis zwischen der Anzahl der Zugriffe pro Adresse ist aber deutlich kleiner (die tatsächlich gemessene Anzahl der Adressen ist in jeder Grafik oberhalb der Messpunkte angegeben. M steht für 10^9).

Abbildung 6.3 zeigt die gleiche Analyse für ein Dreikernsystem. Der gemeinsam genutzte Speicher wird nochmals unterteilt in einen Bereich, der von allen drei Kernen benutzt wurde (blau markiert), und einen, der nur von zwei Kernen angesprochen wird (als 1x2, 1x3 und 2x3 markiert). Dieser letzte Bereich wird im Weiteren als „quergenutzt“ bezeichnet. Wie auch schon auf dem Zweikernsystem sind die von allen drei Kernen gemeinsam genutzten Instruktionen für ca 50 % der Zugriffe zuständig.

Aufgrund dieser Analysen wird eine weitere Möglichkeit zur Leistungssteigerung in Betracht gezogen. Da der überwiegende Teil der Speicherzugriffe im gemeinsam genutzten Instruktionsspeicher angesiedelt ist, und da die Zugriffe einen unmittelbaren Einfluss auf die Laufzeit haben, sollte eine Optimierung des Instruktionsspeichers eine signifikante Verbesserung der Laufzeit bewirken.

6.3 Beschleunigung des Instruktionsspeichers

Das Speichersystem sieht in der Regel nur auf der untersten Ebene des Pufferspeichers eine Aufteilung in Instruktions- und Datenspeicher vor. Daher ist es nicht möglich, sich bei der Optimierung ausschließlich auf den Instruktionsspeicher zu konzentrieren. Bei einem Mehrkernsystem ist es außerdem nicht sinnvoll, den Instruktionsspeicher auf der ersten Ebene für mehr als einen Kern zugänglich zu machen. Die Verteilungslogik würde gerade an dieser Stelle kritische Verzögerungen verursachen.

Daher wurden in dieser Arbeit drei mögliche Ansätze zur Optimierung des Speichers verfolgt. Diese Ansätze sollten eine Verbesserung des Instruktionsspeichers erreichen.

1. Verdoppelung der Kapazität des Pufferspeichers auf der ersten Ebene.

2. Instruktionsspeicher auf der zweiten Ebene mit reduzierter Zugriffszeit.

3. 3D-DRAM-Architektur mit latenzoptimierter Schicht.

Die Zugriffszeit des Instruktionsspeichers auf der ersten Ebene ist bereits so kurz, dass die Daten innerhalb eines Taktes bereitgestellt werden können. Dieses Verhalten entspricht einem idealen Speichermodell. Eine Verdoppelung der Kapazität sollte die Fehlzugriffsrate reduzieren und so das System verbessern.

In Abschnitt 3.3.2 wurde das Pufferspeichermodell beschrieben. In Abbildung 3.17 ist eine separate Instruktionsspeichereinheit angedeutet. Dieses Modell wurde für die folgende Untersuchung verwendet. Neben der Verteilungslogik zwischen der ersten und der zweiten Ebene war eine zusätzliche Stufe zwischen der zweiten Ebene und dem DRAM-Controller erforderlich. Der Instruktionsspeicher auf der zweiten Ebene benötigte nur zwei Zyklen bei einem Treffer und damit weniger als die unspezialisierte Komponente. Als Folge wurde die Kapazität dieses Speichers reduziert. Es wurde nur eine Leseanfrage aus der ersten Ebene zugelassen. Die Verteilung erfolgte anhand der Instruktionsadresse.

Als dritte Möglichkeit wurde die 3D-DRAM-Architektur mit der latenzoptimierten Schicht ohne eine Softwareanpassung evaluiert. Damit wurde die lokalitätsbasierte Verbesserung bewusst herausgehalten, um den alternativen Ansatz überprüfen zu können.

Diese Verbesserungsansätze wurden separat getestet. Die Betriebsfrequenz wurde auf 1000 MHz gesetzt, da die Untersuchungen auf einem Einkernsystem bereits gezeigt haben, dass bei dieser Frequenz der Einfluss des Speichersystems eine signifikante relative Verzögerung verursacht. Als Bezugssystem wurde eine unveränderte 3D-DRAM-Architektur genommen, damit die Laufzeitverbesserungen der gestapelten Technik keinen Einfluss ausüben und die reale Simulationszeit eine überschaubare Spanne

bleibt. Die Modellierung der Pufferspeicherhardware war im Vergleich zu einem Einkernsystem komplexer. Zudem waren Synchronisierungspunkte in der Software notwendig, damit der Pufferspeicher die Daten auch aktualisiert. Diese gesteigerte Komplexität verursachte mehr Fehlerquellen, sodass manche Testfälle nicht abgeschlossen werden konnten. Die Abbildung 6.4 zeigt die gemessene relative Verzögerung, der Pufferspeicher- und der DRAM-Anteil sind zusammengefasst.

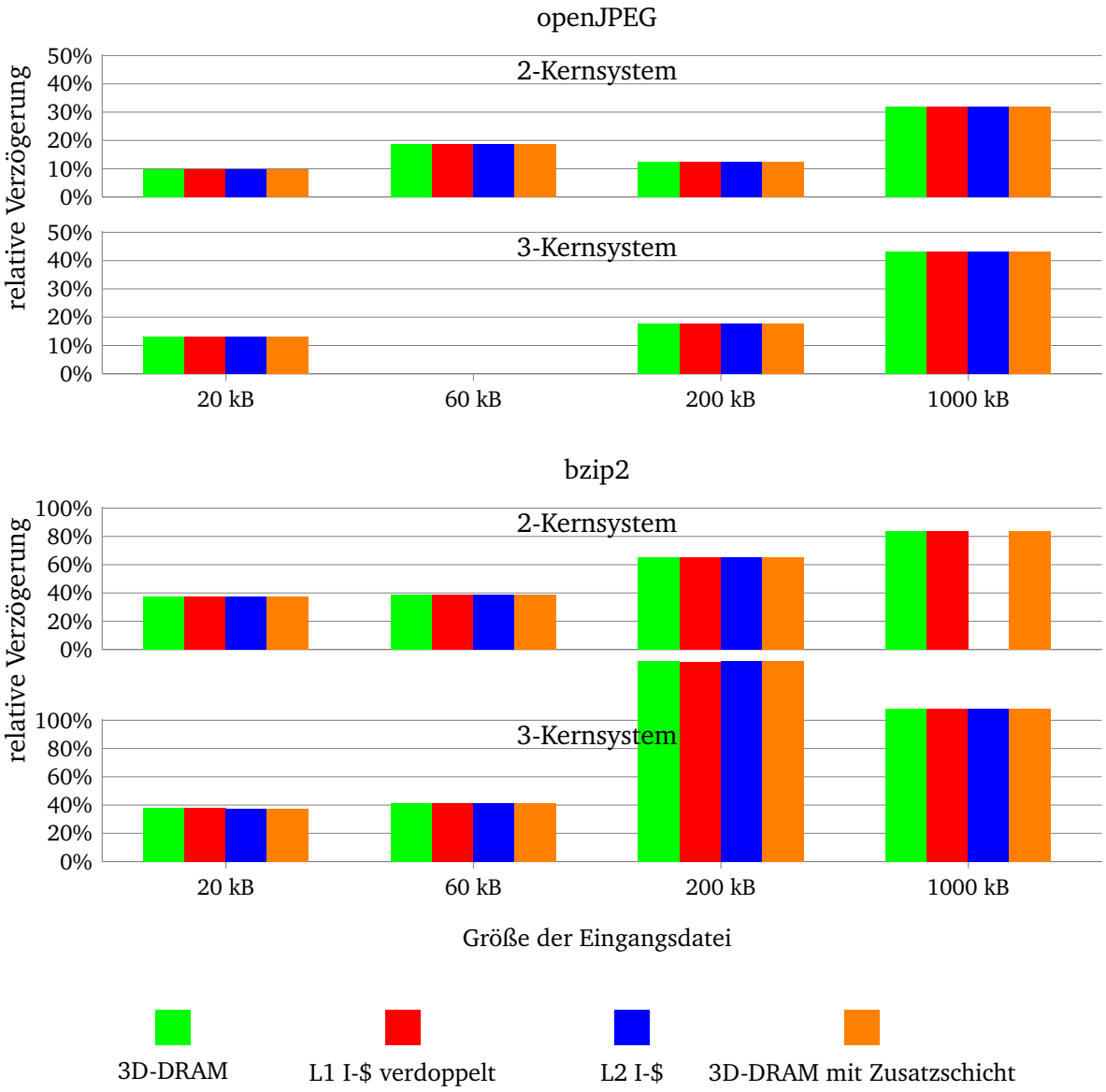


Abbildung 6.4: Vergleich der gesamten relativen Verzögerung des Speichersystems für Zwei- und Dreikernsysteme. Ausgeführt wurden parallelisierte openJPEG- und bzip2-Programme. Die Eingangsdaten wurden in der Größe variiert.

Die fehlenden Werte bei openJPEG und bzip2 zeigen, welche Testfälle fehlerhaft verliefen. Insbesondere für 1000 kB Eingangsdaten war es aufgrund der großen Datenmenge nicht möglich, die Fehlerursache zu finden. Auffällig ist allerdings, dass keiner der untersuchten Optimierungsansätze eine darstellbare Verringerung der Laufzeit bewirken konnte. Die Unterschiede liegen im Bereich von Bruchteilen von Prozent beziehungsweise Promille. Die übliche Größe von 32 kB des Instruktionsspeichers scheint bereits eine optimale Kapazität zu haben.

Zusammenfassend lässt sich feststellen, dass ein Optimierungsansatz, der nur auf der Anzahl der Zugriffe basiert und die Lokalität nicht berücksichtigt, keine signifikante Verbesserung bewirkt.

7 Zusammenfassung und Ausblick

7.1 Memory Wall

Digitale Datenverarbeitungssysteme benötigen zwei Komponenten, um ein Programm ausführen zu können: eine Verarbeitungseinheit und eine Datenaufbewahrungseinheit. Wenn man die Leistungsfähigkeit dieser Komponenten unabhängig voneinander betrachtet, lässt sich eine Differenz von fast fünf Größenordnungen, angewachsen über 25 Jahre, feststellen. Die Leistungsfähigkeit der Speicherkomponenten ist dabei weitgehend mit der Zugriffszeit gleichgesetzt. Die Zuwachsraten der Speichertechnik waren in diesem Zeitraum konstant gering. Jede signifikante Verbesserung der Leistungsfähigkeit des Speichers sollte das gesamte Datenverarbeitungssystem also spürbar verbessern. Genau dieses Potenzial könnte die Entwicklung der Stapeltechnik bringen. Sie erlaubt es nun, Querverbindungen zwischen den Schichten in theoretisch beliebiger Anzahl und an theoretisch beliebiger Stelle zu platzieren. Es bleibt allerdings eine Frage offen: Spielt diese Differenz in der Leistungsfähigkeit in modernen Systemen eine relevante Rolle?

Im Zentrum dieser Arbeit stand die Messung der Leistungsfähigkeit, zurückgeführt auf die Zugriffszeit. Eine umfangreiche Untersuchung des Einflusses des Durchsatzes wurde bereits bei [Woo et al., 2010] durchgeführt. Bei einem Datenverarbeitungssystem spricht man von einer Datenaufbewahrungseinheit. Doch diese Komponente kann ein System aus sehr heterogenen Bestandteilen bilden. Diese Bestandteile bilden eine Speicherhierarchie, wobei die Kapazität und die Zugriffszeit mit jeder Stufe zunehmen. Die Lücke in der Leistungsfähigkeit wird so stufenweise überbrückt. Auf welcher Stufe die Stapeltechnik optimal anzusetzen ist, wird derzeit noch erforscht. Diese Arbeit reiht sich unter einer Vielzahl möglicher Ansätzen ein. Die theoretische Grundlage für die Hypothesen dieser Arbeit bilden das *working set model* von [Denning, 1968] und das Gesetz von [Amdahl, 1967]. Das besser als Lokalitätsprinzip bekannter Zusammenhang besagt, dass 10 % des verwendeten Speichers etwa 90 % aller Instruktionen bereitstellen. Etwas weniger streng gilt dies auch für Daten. Das Gesetz von Amdahl berechnet die Leistungssteigerung für ein Gesamtsystem, wenn nur ein Teilsystem verbessert wurde. Die Kernhypothese dieser Arbeit zufolge bewirkt die Stapeltechnik nur dann eine eine signifikante Steigerung der Leistungsfähigkeit des Gesamtsystems, wenn einerseits der Speicheranteil an der Ausführungszeit groß genug ist und andererseits häufig genutzte Daten durch die Stapeltechnik schneller zur Verfügung gestellt werden.

Neben der genauen Definition von „Ausführungszeit“ muss diese Hypothese auf einem System überprüft werden, das für keine speziellen Aufgaben konzipiert ist, um eine allgemeine Antwort zu erhalten. Ein RISC-Kern, der sowohl allein als auch in einem Mehrkernsystem über einen entsprechenden Pufferspeicher mit einer DRAM-Komponente verbunden ist, wurde für die Testumgebung ausgewählt. Die Bezugsdaten lieferte ein Modell eines DDR2-DRAM von [Micron Technology Inc., 2013]. Das verwendete Modell eines 3D-DRAM beruht auf der Annahme, dass eine Speicherbank auch auf einem gestapelten System auf konventionelle Weise aufgebaut sein muss. Die theoretische Betrachtung der Dynamik eines DRAM-Speichers stützt diese Annahme. Die Architektur des Speichers greift in erste Linie auf Arbeiten von [Weis et al., 2013] und [Zhu et al., 2013] zurück. Des Weiteren wurde eine neue Hardwarearchitektur vorgestellt, die in der Lage ist, häufig genutzte Daten schneller weiterzuleiten.

Zu einem Datenverarbeitungssystem gehören auch Applikationen. Eine Ausführung ist nur sinnvoll, wenn diese Applikationen auch Daten für eine Verarbeitung bekommen. Eine eigens für diese Arbeit entwickelte Analysesoftware - *3DMemory* - wurde vorgestellt. Außerdem wurden die Betriebssystemfunktionalität präsentiert, die erfoderlich ist, um die Applikationen auf dem Testsystem ausführen zu können. Diesen Prozess sowie die Testfallgenerierung automatisierten eine Reihe von Perl-Scripten. Das

Lokalitätsprinzip stellt die theoretische Grunglage dieser Arbeit dar. Zwei gegensätzliche Ausprägungen dieses Prinzips - implementiert als Testfunktionen - bilden den Rahmen für die Einordnung der Messergebnisse.

Die Stapeltechnik wurde bei dieser Arbeit nur für DRAM-Speicher evaluiert, sodass der Einfluss des Pufferspeichers aus den Messwerten herausgerechnet wurde. Die entscheidende Größe war die **relative Verzögerung**. Diese in Prozent angegebene Zahl zeigte an, inwieweit ein System mit DRAM langsamer läuft als mit einem idealen Speicher. Ein System mit einem idealen Speicher stellt momentan nur ein theoretisches Modell dar und bildet somit die Obergrenze für die Leistungsfähigkeit eines realen System. Die relative Verzögerung gibt somit Aufschluss über die potenzielle Leistungssteigerung.

Die Messergebnisse zeigen, dass der DRAM-Speicheranteil an der Ausführungszeit nur für hohe Taktfrequenzen signifikante Größen erreicht. Abhängig von der Applikation beträgt dieser Anteil selbst für die höchste momentan in Serie realisierbare Frequenz - für 4 GHz - zwischen 20 % und 60 %. Nur eine Applikation mit sehr schlechter Datenlokalität erreicht etwa 240 %. Eine Applikation mit zufällig verteilten Daten bildet eine Obergrenze für das Lokalitätsprinzip. Und die Ausführung einer solchen Applikation erreicht eine relative Verzögerung von etwa 300 %. Bezug nehmend auf die Hypothesen der Arbeit lassen sich folgende Aussagen treffen:

Hypothese 1

Wenn alle weiteren Zugriffe auf die Adressen, nach dem ersten Lesen im DRAM, gepuffert sind, kann nur eine Zugriffszeitreduktion des DRAM-Speichers die Ausführungszeit der Applikation verringern.

bestätigt durch den Vergleich von DDR2-DRAM und 3D-DRAM

Der durchgeführte Vergleich zeigte, dass die Reduktion der Gesamtlaufzeit proportional zur Reduktion der Zykluszeit eines DRAM-Zugriffs ist.

Hypothese 2

Je größer die Menge an Adressen ist, die mehr als einmal im DRAM angefragt werden, desto mehr kann die Laufzeit durch Pufferspeicherfunktionalität im DRAM verringert werden.

widerlegt. Es kommt auf die Applikation an

Die Hypothese beruht auf der Annahme, dass jede Applikation eine gewisse Datenlokalität besitzt. Die Untersuchungen zeigten aber, dass das bzip2-Programm in seinem Verhalten einer zufälligen Verteilung von Zugriffsadressen ziemlich nahekommt, sodass der lokalitätsbasierte Ansatz keine messbare Verringerung der Laufzeit erreichen kann.

Hypothese 3

Außer der Implementierung der Pufferspeicherfunktionalität kann kein weiterer Ansatz höhere Leistungssteigerung bewirken.

teilweise bestätigt für Mengenanalyse

Die Analyse der Speichernutzungsdaten auf einem Mehrkernsystem zeigte, dass ungefähr die Hälfte aller Speicherzugriffe in gemeinsam genutzten Instruktionen zu finden sind. Alle betrachteten Optimierungsansätze führten allerdings zu keiner Verbesserung der Laufzeit. Die Hypothese lässt sich nicht abschließend überprüfen, da für ihre Überprüfung Kenntnis von allen möglichen Optimierungsansätzen erforderlich ist.

Die Differenz zwischen der Leistungsfähigkeit der CPU und des DRAM war der Auslöser für die Forschungstätigkeit. Die Ergebnisse der Messungen zeigen allerdings, dass diese Differenz für ein reales

System mit Pufferspeicher bei Weiten nicht das - auf den ersten Blick mögliche - Potenzial für Leistungssteigerung beinhaltet. Der DRAM ist in Bezug auf eine CPU zwar langsam, wird aber auch selten genutzt, sodass sein Anteil an der gesamten Ausführungszeit nur gering ist. Eine Beschleunigung von relativ kleiner Anteile bewirkt eine geringe Beschleunigung des Gesamtsystems. Plakativ gesprochen, reduziert sich die „memory wall“ auf einen „memory stair“, wie in Abbildung 7.1 dargestellt wird.

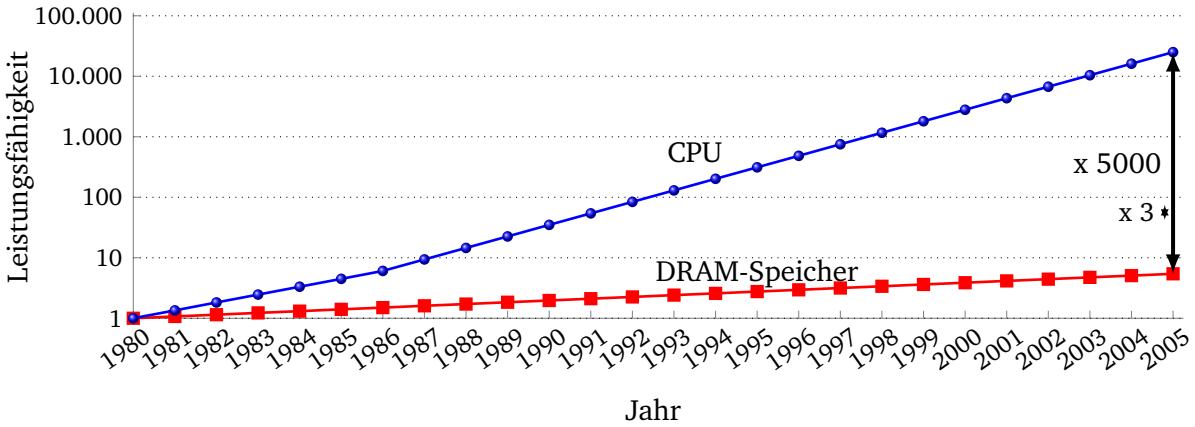


Abbildung 7.1: Direkter Vergleich der Leistungsfähigkeit von CPU und DRAM aus [Hennessy und Patterson, 2012] und die tatsächlich gemessene Lücke in einem Gesamtsystem mit Pufferspeicher. Der Faktor drei ist aus der Messung der Ausführung einer Applikation mit nicht vorhandener Datenlokalität bei 4 GHz Betriebsfrequenz entnommen. Dieser Wert markiert eine Obergrenze für mögliche Leistungssteigerung durch Latenzreduktion im 3D-DRAM.

Die Lücke in der Leistungsfähigkeit ist also nach wie vor vorhanden, doch für ein System mit einem Pufferspeicher spielt sie keine große Rolle. Wenn es also darum geht, die Leistungsfähgkeit eines Systems in Bezug auf den Speicher zu erhöhen, dann erweist sich der Pufferspeicher als eine besser Stelle für Optimierungen. Insbesondere wenn der Pufferspeicher groß genug ist, um alle Daten aufnehmen zu können, stellt sowohl die Latenz als auch die Durchsatzrate eines DRAM für die Gesamtausführungszeit nicht die entscheidende Größe. Wenn zudem die Betriebsfrequenz unter 1 GHz liegt, dann reduziert sich der DRAM-Anteil an der Ausführungszeit auf einen Promillebereich.

7.2 Ausblick

Der Fokus dieser Arbeit lag auf der Latenz. Daher war die Ausführungszeit einer Applikation die entscheidende Messgröße. Die Untersuchungen und vor allem der vorgeschlagene lokalitätsbasierte Ansatz haben gezeigt, dass es durchaus möglich ist, eine signifikante Reduktion der Ausführungszeit zu erreichen und die Leistungsfähigkeit damit zu steigern. Wenn es allerdings allein um die Latenz geht, ist es nicht ausgeschlossen, dass es andere, weniger aufwändige und kostengünstigere Möglichkeiten gibt, zu der erreichten Größenordnung zu gelangen. Die Stapeltechnik bietet also durchaus Möglichkeiten zur Leistungssteigerung. Speziell für DRAM kann es aber nicht zum entscheidenden Kriterium werden. Der Einsatz des Pufferspeichers reduziert die Problematik in diesem Bereich soweit, dass der entsprechende Anteil hinsichtlich des Aufwands zu gering wird. Selbst bei der momentan größten Betriebsfrequenz verzeichnet der DRAM für manche Applikationen eine relative Verzögerung von 20 bis 60 %. Für größere Eingangsdaten sind diese Werte fallend oder gleichbleibend.

Abschließend lässt sich festhalten, dass die (durchaus vorhandene) Vorteile der Stapeltechnik für DRAM nicht primär in der Leistungsfähigkeit zu suchen sind. Unter anderem führt sie zu reduziertem Platz auf der Platine sowie erweitertem Entwurfsraum. Darüber hinaus bietet sie die Möglichkeit, die

einzelnen Komponente besser aufeinander abzustimmen. Wenn es also um DRAM geht, sollte das Gesamtkonzept betrachtet werden, das neben der Hardware auch die Applikation und die Verarbeitungsdaten, sowie weitere Bestandteile des Systems miteinbezieht. Dann kann die gesteigerte Leistungsfähigkeit in Kombination mit anderen Erneuerungen ihren Beitrag zu einem besserem System leisten. Hinsichtlich der Leistungsfähigkeit bietet die Stapeltechnik für DRAM allerdings nur geringe Vorteile.

A Anhang

A.1 Maßstabsgetreue Bilder, die als Eingangsdaten für openJPEG verwendet wurden

siehe nächste Seite ...

1000 kB

200 kB

60 kB

20 kB

Abbildung A.1: Lena

1000 kB

200 kB

60 kB

20 kB

Abbildung A.2: Paprika

1000 kB

200 kB

60 kB

20 kB

Abbildung A.3: Boot

A.2 Häufig genutzten Datenstrukturen, sortiert nach Anzahl der Zugriffe

Applikation	Name	`malloc` oder `calloc` [Datei:Funktion:Zeile]	`free` [Datei:Function:Zeile]
openJPEG	l_t1	t1.c:opj_t1_create:1218	t1.c:opj_t1_destroy:1272
	mqc	mqc.c:opj_mqc_create:363	mqc.c:opj_mqc_destroy:380
	t1->data	t1.c:opj_t1_allocate_buffers:1176	t1.c:opj_t1_destroy:1262
	comp->data	image.c:opj_image_create:67	image.c:opj_image_destroy:90
bzip2	s	bzlib.c:BZ2_bzCompressInit:169	bzlib.c:BZ2_bzCompressEnd:481
	bzf	bzlib.c:BZ2_bzWriteOpen:932	bzlib.c:BZ2_zWriteClose64:1077
	file_stream	stdio.c:fopen:40	stdio.c:fclose:380
	file_stream	stdio.c:fopen:60	stdio.c:fclose:380
XviD	image->y	image.c:image_create:55	image.c:image_destroy:99
	file_stream	stdio.c:fopen:40	stdio.c:fclose:380
	image->u	image.c:image_create:61	image.c:image_destroy:104
	image->v	image.c:image_create:70	image.c:image_destroy:109

A.3 Messwerte

A.3.1 openJPEG

Zyklenzahl					
	20 kB	60 kB	200 kB	1000 kB	DDR2-Fehler
Lena	31749504	111540688	414321699	1623932231	1338822982
Pepper	36246798	123717453	444170988	1796186645	1345999248
Sailboat	34375849	123573580	460287720	1874750034	1338942696

Laufzeit mit Pufferspeicher [ns]						
		20 kB	60 kB	200 kB	1000 kB	DDR2-Fehler
512 kB L2	Lena	307458068	1276977844	4118889724	19260018892	15892599924
	Pepper	350375084	1415291428	4401987372	21128553660	15889042924
	Sailboat	333379564	1425056596	4574708948	22110705812	15862732364
256 kB L2	Lena	307555988	1277259668	4123419612	19286807060	15919335340
	Pepper	350475548	1415581108	4406523636	21155348156	15915784012
	Sailboat	333479452	1425346548	4579258516	22137497660	15889461396
128 kB L2	Lena	307615476	1277745796	4148454036	19308633220	15938387476
	Pepper	350537260	1416065372	4431561356	21178795596	15934836188
	Sailboat	333538892	1425828484	4604292868	22161415428	15908282044
64 kB L2	Lena	307719884	1279077388	4158285676	19352558092	
	Pepper	350649036	1417402700	4441788716	21223391748	
	Sailboat	333647652	1427164924	4614744556	22206433676	
32 kB L2	Lena	307846740	1286521404	4170269084	19353859396	
	Pepper	350776044	1424948972	4453838452	21224836932	
	Sailboat	333773236	1434791164	4626802468	22207888700	

Laufzeit mit DDR2 DRAM [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	Lena	308427586500	1279264274500	4136965578500	16462313122500
	Pepper	351355098500	1417616898500	4420192266500	16457993074500
	Sailboat	334357578500	1427404658500	4593060170500	16431378218500
256 kB L2	Lena	308942802500	1280784154500	4163008906500	17180865122500
	Pepper	351883274500	1419180674500	4446268146500	17180863242500
	Sailboat	334883298500	1428968154500	4619216818500	17180864114500
128 kB L2	Lena	309257386500	1283474842500	4302809970500	16718717554500
	Pepper	352207954500	1421860250500	4586089746500	16714494042500
	Sailboat	335195650500	1431637154500	4759018546500	16686519906500

Laufzeit mit 3D-DRAM bei 125 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	Lena	307661540000	1277462316000	4122596356000	19384158668000
	Pepper	350581132000	1415784252000	4405713564000	21252855172000
	Sailboat	333585052000	1427005020000	4578473860000	22235173908000
256 kB L2	Lena	307834636000	1277977636000	4131575764000	19437716748000
	Pepper	350758516000	1416314740000	4414709140000	21306437388000
	Sailboat	333761460000	1427005020000	4587488092000	22288741348000
128 kB L2	Lena	307937828000	1278904460000	4181636644000	19483527956000
	Pepper	350866116000	1417238948000	4464780804000	21355730452000
	Sailboat	333864636000	1427005020000	4637553660000	22339040748000
64 kB L2	Lena	308137756000	1281517940000	4201880156000	19578068596000
	Pepper	351081332000	1419863884000	4485840236000	21451663700000
	Sailboat	334073452000	1427005020000	4659079092000	22435826652000
32 kB L2	Lena	308383492000	1296538196000	4227416292000	19580698164000
	Pepper	351327820000	1435093348000	4511544708000	21454580308000
	Sailboat	334316788000	1427005020000	4684798732000	22438812740000

Laufzeit mit 3D-DRAM bei 1000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	Lena	38621862500	160152230500	519514702500	2551818702500
	Pepper	43988881500	177452244500	554933784500	2785587724500
	Sailboat	41863926500	178679026500	576560953500	2908540043500
256 kB L2	Lena	38755419500	160529513500	525309895500	2585936413500
	Pepper	44125980500	177839713500	560737262500	2819703703500
	Sailboat	42000220500	179066841500	582390005500	2942648991500
128 kB L2	Lena	38837204500	161165554500	557152804500	2612909263500
	Pepper	44210534500	178472973500	592588847500	2848673868500
	Sailboat	42081838500	179697054500	614224802500	2972198513500
64 kB L2	Lena	38973966500	162886442500	569464460500	2666272254500
	Pepper	44356512500	180193437500	605372683500	2902827154500
	Sailboat	42224141500	181416587500	627298293500	3026872759500
32 kB L2	Lena	39138600500	172302702500	584101586500	2667888537500
	Pepper	44521037500	189750385500	620111857500	2904667268500
	Sailboat	42387053500	191067494500	642041227500	3028715042500

Laufzeit mit 3D-DRAM bei 4000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	Lena	9794971375	40427957875	133301723125	744116525625
	Pepper	11138485625	44760917375	142179940875	802707497125
	Sailboat	10606863125	45072127375	147613930125	833580409125
256 kB L2	Lena	9917607125	40773885875	138596826375	775252216875
	Pepper	11264378375	45116161875	147482622875	833843049875
	Sailboat	10732021625	45427687875	152933069375	864712855375
128 kB L2	Lena	9992751375	41356056125	167678676125	799825539375
	Pepper	11342050875	45695762875	176568221375	860212370625
	Sailboat	10807009375	46004500125	182014456125	891609476375
64 kB L2	Lena	10117937125	42922193875	178905424125	848356686625
	Pepper	11475611125	47268484625	188235871875	909516857125
	Sailboat	10937252375	47576377125	193931273625	941386980125
32 kB L2	Lena	10268518375	51525893125	192231820125	849878977375
	Pepper	11626099625	55987167375	201637721625	911185187125
	Sailboat	11086255625	56387151625	207347973375	943068923875

Laufzeit mit 3D-DRAM und latenzoptimierter Schicht bei 4000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	Lena	9777292625	40408528875	133283653375	744101495875
	Pepper	11120832875	44741488375	142159419875	802692467375
	Sailboat	10589210375	45052697375	147593407875	833575344625
256 kB L2	Lena	9898022125	40753650875	138576835125	775242576875
	Pepper	11244819375	45095926875	147460723875	833841023125
	Sailboat	10712461375	45407451875	152911065125	864702819375
128 kB L2	Lena	9973010375	41335301125	167653267375	799731241125
	Pepper	11322335875	45674929875	176542726375	860104538125
	Sailboat	10787293125	45983640125	181989384125	
64 kB L2	Lena	10097338125	42897798875	178832634125	848233646125
	Pepper	11454994375	47243985625	188155774875	909382677375
	Sailboat	10916626125	47552007125	193628919875	941239118875
32 kB L2	Lena	10246913625	51479294375	192133873375	849729200625
	Pepper	11604442875	55939138625	201535605125	911015961875
	Sailboat	11064675625	56338237875	207251102875	942900387875

Zyklenzahl mit lokalitätsbasierter Software				
	20 kB	60 kB	200 kB	1000 kB
Lena	31749498	111540660	414321671	1623932203
Pepper	36246792	123717425	444170960	1796186617
Sailboat	34375843	123573552	460287692	1874750006

Laufzeit mit Pufferspeicher und lokalitätsbasierter Software [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	Lena	9987323625	36195870875	134789876875	545029106375
	Pepper	11398421125	40060190375	144015407375	597560975625
	Sailboat	10849890875	40250106625	149925039125	624295275875
256 kB L2	Lena	9985464125	36180758625	134016726625	544383988125
	Pepper	11396505875	40045182625	143242212875	596867034125
	Sailboat	10848025625	40235101375	149151982625	623581706875
128 kB L2	Lena	9987323625	36195870875	134789876875	545029106375
	Pepper	11398421125	40060190375	144015407375	597560975625
	Sailboat	10849890875	40250106625	149925039125	624295275875
64 kB L2	Lena	9990641625	36237198125	135088858875	546371817375
	Pepper	11401933875	40101687375	144325970125	598920941875
	Sailboat	10853314875	40291590875	150243519375	625663002125
32 kB L2	Lena	9994525375	36458245125	135453797125	546407309625
	Pepper	11405876625	40326025375	144692374625	598960836625
	Sailboat	10857195125	40517619125	150609113875	625663002125

Laufzeit mit 3D-DRAM und lokalitätsbasiertem Ansatz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	Lena	10141399375	36583007375	136643014625	615921993625
	Pepper	11554663625	40458013625	145897163375	668600059375
	Sailboat	11005896875	40653831875	151844986125	695505657125
256 kB L2	Lena	10243104375	36811098375	139440672375	631822814625
	Pepper	11659783625	40690991625	148703236625	684501660875
	Sailboat	11110175875	40884184375	154667150125	711399101875
128 kB L2	Lena	10309566875	37148160875	154150718625	644131033625
	Pepper	11727026625	41029546125	163413773125	697686290125
	Sailboat	11175913625	41222971625	169373320375	725079463125
64 kB L2	Lena	10406080875	38000511875	160153403625	669116866125
	Pepper	11831876375	41888242125	169655757375	722969193125
	Sailboat	11277108875	42081504375	175781268125	750352206625
32 kB L2	Lena	10505986875	42381552375	167088302625	668295267375
	Pepper	11933763625	46346795875	176642436625	722799510875
	Sailboat	11376880375	46587130375	182764235625	750393237125

A.3.2 bzip2

Zyklenzahl				
	20 kB	60 kB	200 kB	1000 kB
kernel	13295389	32500178	134283776	475227222
pride	19923079	41641901	141267052	561539642
robot	1766580	41567192	142387585	522089193

Laufzeit mit Pufferspeicher [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	kernel	190581460000	456388740000	1895866244000	6648240332000
	pride	259427836000	553843972000	1953849940000	7471600852000
	robot	22032460000	562032452000	1932673140000	7067971132000
256 kB L2	kernel	194917516000	461517956000	1914741660000	6714342436000
	pride	266794388000	562130788000	1974781444000	7538068820000
	robot	22032460000	571396188000	1959019916000	7147099308000
128 kB L2	kernel	200414004000	468566940000	1938581332000	6792973036000
	pride	269930652000	566090956000	1994206668000	7610919692000
	robot	22032460000	574579876000	1969171076000	7189048900000
64 kB L2	kernel	200824108000	471236852000	1954083132000	6847945308000
	pride	270471660000	568175236000	2009013060000	7668318988000
	robot	22032508000	576904828000	1976717932000	7213934028000
32 kB L2	kernel	201953596000	475128380000	1970228244000	6903240540000
	pride	271207116000	571758668000	2023391580000	7721913636000
	robot	22034524000	579272588000	1983959244000	7235568292000

Laufzeit mit DDR2 DRAM [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	kernel	191567250500	460761090500	1925061770500	6752304650500
	pride	260459930500	558252762500	1978787682500	7568634338500
	robot	22282194500	566328690500	1951079842500	7139227954500
256 kB L2	kernel	215881218500	489570954500	2031691786500	7125819754500
	pride	301847730500	604828746500	2096975274500	7944500818500
	robot	22282194500	618958050500	2099380282500	7585000674500
128 kB L2	kernel	246908466500	529535018500	2167357610500	7573955714500
	pride	319586322500	627320402500	2207251690500	8358642434500
	robot	22282194500	637005210500	2156922370500	7822715458500

Laufzeit mit 3D-DRAM bei 125 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	kernel	190790116000	457563428000	1901951700000	8874023140000
	pride	259649148000	554757660000	1960736452000	7498598132000
	robot	22059340000	562934620000	1936539468000	7082921148000
256 kB L2	kernel	199276092000	469105396000	1939366852000	14304003876000
	pride	274090564000	571094108000	2007722452000	7647711564000
	robot	22059340000	581438932000	1988743380000	7239623004000
128 kB L2	kernel	210396772000	484825692000	1987444580000	14478675588000
	pride	280555220000	579176276000	2051044108000	7810251148000
	robot	22059340000	587861076000	2009366140000	7324772068000
64 kB L2	kernel	211198948000	490785916000	2019100644000	14601460348000
	pride	281615612000	583295340000	2083772964000	7937018716000
	robot	22059444000	592418908000	2024079196000	7373434060000
32 kB L2	kernel	213406972000	499480300000	2051247684000	14725552540000
	pride	283056068000	590464964000	2115928668000	8056632332000
	robot	22062996000	597005748000	2038117076000	7415651428000

Laufzeit mit 3D-DRAM bei 1000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	kernel	24016445500	58141074500	244487307500	858505377500
	pride	32633994500	70310911500	250617387500	958983732500
	robot	2779028500	71313775500	246315373500	902015851500
256 kB L2	kernel	29605988500	64731876500	268638438500	942939864500
	pride	42119161500	80946587500	277416160500	1044009843500
	robot	2779028500	83309290500	280026451500	1003292957500
128 kB L2	kernel	36546569500	73647528500	298821772500	1042530560500
	pride	46035182500	85924216500	302014031500	1136377057500
	robot	2779028500	87340907500	292833620500	1056233198500
64 kB L2	kernel	37075140500	77040285500	318302321500	1111549108500
	pride	46731280500	88596925500	320512036500	1207942108500
	robot	2779086500	90337294500	302584506500	1088317178500
32 kB L2	kernel	38534266500	81987967500	338894314500	1182082684500
	pride	47678976500	93154518500	338785538500	1275865313500
	robot	2781834500	93405956500	311966092500	1116221679500

Laufzeit mit 3D-DRAM bei 4000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	kernel	6146640625	15339256125	66641617625	234835747125
	pride	8309672375	18372457875	67346714125	258153953125
	robot	713126875	18607950375	65059029375	239126866875
256 kB L2	kernel	11256877375	21363145625	88701226875	311949002625
	pride	16980700875	28092785875	91827164125	335829620875
	robot	713126875	29569484625	95853724875	331648724625
128 kB L2	kernel	17591967625	29502707375	116254732875	402870928125
	pride	20551870375	32634496125	114282724875	420157927375
	robot	713126875	33250222375	107535636125	379946888625
64 kB L2	kernel	18075207125	32599687875	134031551125	465840409125
	pride	21188184375	35077014375	131162398375	485404836375
	robot	713179625	35989619625	116451585375	409291614125
32 kB L2	kernel	19409434875	37119137875	152845724875	530278934125
	pride	22054588625	39239386625	147846987125	547437119875
	robot	715702625	38796313375	125031985125	434818448625

Laufzeit mit 3D-DRAM und latenzoptimierter Schicht bei 4000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	kernel	6138866625	15331001625	66631659625	234834510875
	pride	8301872375	18364111875	67337426625	258153472875
	robot	708550875	18599526375	65049643375	239125702375
256 kB L2	kernel	11248245375	21354302625	88695816625	311942584125
	pride	16971548875	28083399875	91820953125	335826036875
	robot	708550875	29559968625	95841738875	331650574625
128 kB L2	kernel	17583257625	29492718125	116247388125	402867180375
	pride	20542640375	32624148125	114276702875	420155514875
	robot	708550875	33240394375	107530939125	379949053375
64 kB L2	kernel	18066237125	32589054375	134022911875	465839510875
	pride	21178798375	35066146375	131155179875	485402357375
	robot	708603625	35979323625	116445605625	409291484625
32 kB L2	kernel	19399866875	37108420375	152836022375	530277289125
	pride	22044916625	39228414625	147839416625	547436435875
	robot	710918625	38785705375	125025737875	434816560125

Zyklenzahl mit lokalitätsbasierter Software				
	20 kB	60 kB	200 kB	1000 kB
kernel	13295302	32500091	134283689	475227135
pride	19922992	41641814	141266965	561539555
robot	1766493	41567105	142387498	522089106

Laufzeit mit Pufferspeicher und lokalitätsbasierter Software [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	kernel	6217418375	14140340875	58476775125	205116256125
	pride	8391514375	17479703875	60152158125	230314171375
	robot	743081875	17701180125	60527180625	221145265125
256 kB L2	kernel	6048979875	14300929125	59066023125	207182076125
	pride	8294552375	17364619625	60811623125	232406132625
	robot	743204125	17609358125	60230929875	219917620375
128 kB L2	kernel	6218649875	14522298625	59808702875	209640297875
	pride	8394790875	17485987375	61410684625	234652525875
	robot	743204125	17707348625	60546430125	221207954875
64 kB L2	kernel	6232226125	14608153375	60304727625	211398546125
	pride	8411580875	17552050375	61880852625	236470470875
	robot	743208375	17782617375	60784551375	222001753625
32 kB L2	kernel	6269689625	14735541375	60838023375	213227038125
	pride	8435046375	17669136875	62358239125	238239318375
	robot	743250375	17856957625	61015345625	222679311125

Laufzeit mit 3D-DRAM und lokalitätsbasiertem Ansatz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	kernel	6075239875	15224236375	66107751625	232564225625
	pride	8252349375	18163042625	66554709375	255743811875
	robot	762298375	18356288375	63966756875	235592061375
256 kB L2	kernel	11297983875	21244431125	88109537875	720416574875
	pride	16857574125	28021550875	91229107375	333906247375
	robot	762298375	29382570125	94933411125	329072852125
128 kB L2	kernel	17553965625	29433079625	115613628875	400662618625
	pride	20516333875	32467164375	113386454625	417106473625
	robot	762298375	33003950375	106559569625	376646882375
64 kB L2	kernel	18064449375	32618112875	133802901875	465062913625
	pride	21141982125	34932586875	130522955625	483192558625
	robot	762456375	35835653125	115531495375	406479503625
32 kB L2	kernel	19480955625	37355734875	153697489875	533272765375
	pride	22017628125	39278711875	148237090625	548732408625
	robot	764562875	38650597375	124246842125	431981587625

A.3.3 XviD

Zyklenzahl				
	20 kB	60 kB	200 kB	1000 kB
calen	39795488	77380181	244510539	1324641231
princ	33728937	66612997	205317023	955289160
traff	32439913	61515229	184895752	1102336033

Laufzeit mit Pufferspeicher [ns]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	calen	356468620	685035764	2147098812	11614841676
	princ	305363500	594058028	1821526084	8504817284
	traff	294583284	552413492	1649907380	9759233348
256 kB L2	calen	356704084	685531980	2148879972	11624985028
	princ	305587284	594542564	1823259708	8514363980
	traff	294827980	552920988	1651739300	9769937548
128 kB L2	calen	356983276	686101084	2150708308	11634511220
	princ	305862388	595109156	1825063668	8523591604
	traff	295109684	553508404	1653375468	9779085172
64 kB L2	calen	357175676	686424516	2151623788	11639371332
	princ	306060156	595447476	1826080220	8528538228
	traff	295293204	553838124	1654360740	9784201276
32 kB L2	calen	357288964	686632988	2152195404	11642431436
	princ	306153836	595631940	1826552612	8530779956
	traff	295390068	554006028	1654794052	9786400348

Laufzeit mit DDR2 DRAM [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	calen	361172882500	690452242500	2154976826500	11638831322500
	princ	310053698500	599479354500	1829478994500	8528167634500
	traff	299268250500	557801914500	1657811826500	9783003874500
256 kB L2	calen	362512722500	693282762500	2165166618500	11696965666500
	princ	311327554500	602249786500	1839416530500	8582916274500
	traff	300661658500	560699674500	1668311050500	9844423898500
128 kB L2	calen	364113282500	696569682500	2175776538500	11752474794500
	princ	312899706500	605523818500	1849889082500	8636684522500
	traff	302281402500	564100666500	1677803874500	9897523538500

Laufzeit mit 3D-DRAM bei 125 MHz [ns]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	calen	357447508	686166524	2148754548	11619944116
	princ	306339212	595189676	1823197716	8509784012
	traff	295558028	553538420	1651567868	9764290660
256 kB L2	calen	357896084	687143292	2152349036	11640500164
	princ	306764212	596144020	1826676652	8529098620
	traff	296026172	554541044	1655251332	9785955540
128 kB L2	calen	358453364	688302364	2156107324	11660283076
	princ	307311324	597296500	1830406516	8548291588
	traff	296588468	555740724	1658626500	9804895444
64 kB L2	calen	358848604	688974676	2158020548	11670520796
	princ	307721044	598003860	1832556868	8558693604
	traff	296969756	556432060	1660692700	9815674124
32 kB L2	calen	359082828	689410748	2159238300	11677079220
	princ	307909692	598385052	1833546812	8563496396
	traff	297164612	556773572	1661606788	9820352468

Laufzeit mit 3D-DRAM bei 1000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	calen	45739962500	86995594500	270390278500	1458027577500
	princ	39347721500	75624455500	229711238500	1069113164500
	traff	37999105500	70409897500	208248778500	1226045092500
256 kB L2	calen	46051422500	87641153500	272687907500	1471079974500
	princ	39644632500	76256161500	231956385500	1081435138500
	traff	38323077500	71070412500	210622055500	1239856625500
128 kB L2	calen	46410124500	88363707500	274996921500	1483015548500
	princ	39998643500	76975637500	234222869500	1092998725500
	traff	38684420500	71816004500	212679557500	1251336361500
64 kB L2	calen	46650908500	88766240500	276128364500	1489023155500
	princ	40246194500	77396544500	235484187500	1099117550500
	traff	38914583500	72226651500	213907984500	1257666767500
32 kB L2	calen	46793346500	89026481500	276833109500	1492787777500
	princ	40365067500	77627794500	236074286500	1101913714500
	traff	39037152500	72437425500	214441228500	1260390606500

Laufzeit mit 3D-DRAM bei 4000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	calen	12304155625	22753971875	69071249625	369051912625
	princ	10703067625	19911989125	58914427125	271704319625
	traff	10365116375	18601733625	53541274625	311023896875
256 kB L2	calen	12589516125	23344699375	71172595625	380984092125
	princ	10975164125	20490190625	60968539625	282961222125
	traff	10662050125	19206284625	55712837375	323662041625
128 kB L2	calen	12917289125	24004300625	73271634875	391854528875
	princ	11298698125	21147017125	63035675875	293511823375
	traff	10992182125	19886844125	57589449375	334108399375
64 kB L2	calen	13136882625	24371297875	74305519125	397337112125
	princ	11524453625	21530658875	64180491125	299090721125
	traff	11202059625	20261233375	58701957875	339876448625
32 kB L2	calen	13266881375	24608753875	74951034625	400783192125
	princ	11633054625	21741717875	64716475875	301635891125
	traff	11314098625	20453787375	59195976625	342372894875

Laufzeit mit 3D-DRAM und latenzoptimierter Schicht bei 4000 MHz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	calen	12076975875	22520075375	68802046125	368548053375
	princ	10476336875	19676978625	58639414375	271220041875
	traff	10138846625	18366667125	53273631125	310555712625
256 kB L2	calen	12336353875	23048544625	70658412375	379006962375
	princ	10724079125	20194519875	60441622375	281035146375
	traff	10406991875	18904668875	55184195625	321582224375
128 kB L2	calen	12626530875	23647667875	72608221625	389151130875
	princ	11010747875	20788775375	62345237125	290773886125
	traff	10699544375	19523134125	56905464625	331366263125
64 kB L2	calen	12822902875	23985026875	73587302625	394329255875
	princ	11214321875	21141969375	63423278125	296096741875
	traff	10887347625	19869917125	57964936125	336912836875
32 kB L2	calen	12935764625	24190644625	74141531875	397254767625
	princ	11307960375	21324779875	63888600125	298331767875
	traff	10987917625	20045380125	58411952125	339160087625

Zyklenzahl mit lokalitätsbasierter Software				
	20 kB	60 kB	200 kB	1000 kB
calen	39794663	77379356	244509714	1324640406
princ	33728112	66612172	205316198	955288335
traff	32439088	61514404	184894927	1102335208

Laufzeit mit Pufferspeicher und lokalitätsbasierter Software [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	calen	11172036625	21469652375	67290897625	364014837375
	princ	9568343625	18614761375	57075287375	266537445375
	traff	9231818875	17311017625	51699346125	305763413125
256 kB L2	calen	11169883375	21463560125	67268993375	363893218625
	princ	9565488125	18607505125	57047730625	266368928625
	traff	9228928375	17302007125	51674305375	305613087125
128 kB L2	calen	11172036625	21469652375	67290897625	364014837375
	princ	9568343625	18614761375	57075287375	266537445375
	traff	9231818875	17311017625	51699346125	305763413125
64 kB L2	calen	11178121125	21479778125	67319305875	364164007875
	princ	9574690125	18625548625	57106689625	266689133625
	traff	9237709375	17321274125	51730078625	305920874625
32 kB L2	calen	11181651125	21486260375	67336891875	364258829375
	princ	9577515125	18631162375	57121472875	266758890875
	traff	9240702875	17326495875	51743236375	305987531125

Laufzeit mit 3D-DRAM und lokalitätsbasiertem Ansatz [ps]					
		20 kB	60 kB	200 kB	1000 kB
512 kB L2	calen	11750421125	22140597375	68265088875	367001449875
	princ	10145089125	19287263375	58063825125	269458967875
	traff	9807389875	17977680375	52676960375	308664849375
256 kB L2	calen	11917385625	22477310125	69464844625	373713090875
	princ	10299864875	19614627375	59233564875	275901198375
	traff	9978442125	18322165125	53906856375	315822274125
128 kB L2	calen	12098886375	22850851625	70639381375	379942844625
	princ	10479187625	19980003625	60380572875	281762676125
	traff	10158689625	18699308875	54951733625	321665346375
64 kB L2	calen	12229015125	23070338125	71265268125	383213917625
	princ	10615465625	20213459125	61057286125	285076175125
	traff	10286191375	18921378875	55617584625	325103980625
32 kB L2	calen	12309684625	23213798125	71642531125	385187953125
	princ	10682515375	20338851625	61375302375	286592103875
	traff	10356072875	19038883875	55903453375	326526058875

Literaturverzeichnis

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA. ACM.

[Austin et al., 2002] Austin, T., Larson, E., und Ernst, D. (2002). Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67.

[Belady, 1966] Belady, L. (1966). A study of replacement algorithms for virtual storage computers. *IBM Systems J.*, 5,2:78–101.

[Berg und Hagersten, 2004] Berg, E. und Hagersten, E. (2004). Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 20–27. IEEE.

[Bied, 2014] Bied, M. (2014). Development and Integration of a Hardware Oriented NoC Transfer Protocol Specialized for Software Parallelization. Technical report, TU Darmstadt, Integrierte Elektronische Systeme.

[Bridges et al., 2007] Bridges, M. J., Vachharajani, N., Zhang, Y., Jablin, T. B., und August, D. I. (2007). Revisiting the sequential programming model for multi-core. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*, pages 69–84.

[Chehab, 2015] Chehab, L. S. (2015). Formal Verification of a MIPS I Specified Processor Core Implementation. Technical report, TU Darmstadt, Integrierte Elektronische Systeme.

[Chen et al., 2012] Chen, K., Li, S., Muralimanohar, N., Ahn, J. H., Brockman, J. B., und Jouppi, N. P. (2012). Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 33–38. EDA Consortium.

[Dahlem, 2015] Dahlem, J. (2015). Portierung und Parallelisierung des Bzip2- und Xvid-Programms auf einem MIPS I System. Technical report, TU Darmstadt, Integrierte Elektronische Systeme.

[Denning, 1968] Denning, P. J. (1968). The working set model for program behavior. *Commun. ACM*, 11(5):323–333.

[Dong et al., 2010] Dong, X., Xie, Y., Muralimanohar, N., und Jouppi, N. P. (2010). Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society.

[DRAMPower, 2012] DRAMPower (2012). TU DELFT, TU EINDHOVEN, TU KAISERSLAUTERN. http://www.es.ele.tue.nl/drampower/.

[Ferrari, 1978] Ferrari, D. (1978). *Computer Systems Performance Evaluation* -. Prentice-Hall, New York.

[Ford, 1977] Ford, B. (1977). Preparing conventions for parameters for transportable. In *Portability of Numerical Software, Workshop*, pages 68–91, London, UK, UK. Springer-Verlag.

[Ganzhorn und Walter, 1975] Ganzhorn, K. und Walter, W. (1975). *Die geschichtliche Entwicklung der Datenverarbeitung* -. IBM Deutschland GmbH.

[Gupta et al., 2013] Gupta, S., Xiang, P., Yang, Y., und Zhou, H. (2013). Locality principle revisited: A probability-based quantitative approach. *Journal of Parallel and Distributed Computing*, 73(7):1011–1027.

[Hennessy und Patterson, 2012] Hennessy, J. L. und Patterson, D. A. (2012). *Computer architecture: a quantitative approach*. Elsevier.

[Hilbert und López, 2011] Hilbert, M. und López, P. (2011). The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(6025):60–65.

[Hook und Gherman, 2006] Hook, B. und Gherman, D. (2006). *Portabler Code - Einführung in die plattformunabhängige Softwareentwicklung*. Open Source Press, München, 1. aufl. edition.

[Ismail et al., 2013] Ismail, M. A., Altaf, T., und Mirza, S. H. (2013). Mcsmc: A new parallel multi-level cache simulator for multi-core processors. In *Electronics, Communications and Photonics Conference (SIECPC), 2013 Saudi International*, pages 1–6. IEEE.

[Jacob, 2006] Jacob, B. (2006). Dramsim version 2. http://www.eng.umd.edu/ blj/dramsim/v1/.

[Jacob et al., 2007] Jacob, B., Ng, S., und Wang, D. (2007). *Memory Systems - Cache, DRAM, Disk*. Morgan Kaufmann, San Francisco, Calif, 1. aufl. edition.

[Jain, 1991] Jain, R. (1991). *The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling*. John Wiley and Sons, Inc., New York USA.

[Jeff Gilchrist, 2015] Jeff Gilchrist (2015). pbzip2. compression.ca/pbzip2.

[Jessen, 1996] Jessen, E. (1996). Die entwicklung des virtuellen speichers. *Informatik-Spektrum*, 19(4):216–219.

[JPEG, 2013] JPEG (2013). Jpeg2000. jpeg.org.jpeg2000.

[Julian Seward, 2015] Julian Seward (2015). bzip2. http://www.bzip.org. v1.06.

[Jun et al., 2012] Jun, M., Kim, M.-J., und Chung, E.-Y. (2012). Asymmetric dram synthesis for heterogeneous chip multiprocessors in 3d-stacked architecture. In *Proceedings of the International Conference on Computer-Aided Design*, pages 73–80. ACM.

[Karkowski und Corporaal, 1997] Karkowski, I. und Corporaal, H. (1997). Overcoming the limitations of the traditional loop parallelization. In *High-Performance Computing and Networking*, pages 898–907. Springer.

[Keeth et al., 2008] Keeth, B., Baker, R. J., Johnson, B., und Lin, F. (2008). *DRAM Circuit Design - Fundamental and High-Speed Topics*. John Wiley & Sons, New York, 2. auflage edition.

[Lee et al., 2015] Lee, Y., Kim, J., Jang, H., Yang, H., Kim, J., Jeong, J., und Lee, J. W. (2015). A fully associative, tagless dram cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 211–222. ACM.

[Li et al., 2011] Li, S., Chen, K., Ahn, J. H., Brockman, J. B., und Jouppi, N. P. (2011). Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 694–701. IEEE.

[Lienig et al., 2012] Lienig, J., Dietrich, M. E., Lienig, J., und Dietrich, M. (2012). *Entwurf integrierter 3D-Systeme der Elektronik* -. Springer-Verlag, Berlin Heidelberg New York, 1. aufl. edition.

[Liu et al., 2005] Liu, C. C., Ganusov, I., Burtscher, M., und Tiwari, S. (2005). Bridging the processor-memory performance gap with 3d ic technology. *Design & Test of Computers, IEEE*, 22(6):556–564.

[Loh, 2008] Loh, G. H. (2008). 3d-stacked memory architectures for multi-core processors. *SIGARCH Comput. Archit. News*, 36(3):453–464.

[Matsumoto und Nishimura, 1998] Matsumoto, M. und Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30.

[Mentor Graphics Corporation, 2013] Mentor Graphics Corporation (2013). *Modelsim SE Users Manual*.

[Micron Technology, Inc., 2006] Micron Technology, Inc. (2006). *DDR3 SDRAM MT41JXXXMX*. Micron Technology, Inc., Boise, USA. Rev. N 11/14 EN.

[Micron Technology Inc., 2013] Micron Technology Inc. (2013). DDR2 DRAM model. http://www.micron.com/products/dram. Last accessed on Jul 3, 2013.

[Microsoft, 2014] Microsoft (2014). Windows bitmap. https://msdn.microsoft.com.

[MIPS Technologies Inc., 2013] MIPS Technologies Inc. (2013). *MIPS Architecture For Programmers*.

[MoMuSys, 1997] MoMuSys (1997). Momusys: Modile multimedia system. http://cordis.europa.eu/infowin/acts/analysys/products/thematic/mpeg4/momusys/momusys.htm. archiviert am 17.09.2009.

[Moore, 1965] Moore, G. E. (1965). Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117.

[MPEG, 2015] MPEG (2015). Mpeg-iv. mpeg.chiariglione.org.

[Newmann, 1945] Newmann, J. v. (1945). First Draft of a Report on the EDVAC. Technical report, University of Pennsylvania, Moore School of Electrical Engineering.

[OneSpin Solutions, 2014] OneSpin Solutions (2014). Onespin 360. http://www.onespin.com. Version 2014_12(84).

[OpenJPEG, 2014] OpenJPEG (2014). Openjpeg. http://www.openjpeg.org. v2997.

[Padua et al., 1980] Padua, D. A., Kuck, D. J., und Lawrie, D. H. (1980). High-speed multiprocessors and compilation techniques. *IEEE Trans. Computers*, 29(9):763–776.

[Rauber und Rünger, 2008] Rauber, T. und Rünger, G. (2008). *Multicore: parallele Programmierung*. Informatik im Fokus. Springer, Berlin.

[Rausch, 2015] Rausch, S. (2015). Portierung einer C-Implementierung des x.264 Algorithmus auf ein MIPS I System. Technical report, TU Darmstadt, Integrierte Elektronische Systeme.

[Reuter, 2015] Reuter, M. (2015). Implementation and Integration of a Simulative FPU Model in a MIPS-I Core. Technical report, TU Darmstadt, Integrierte Elektronische Systeme.

[Rhoads, 2013] Rhoads, S. (2013). plasma. opencores.org/project,plasma. Last accessed on Jul 30, 2013.

[Schneider, 1980] Schneider, H. J. (1980). Tagung des German Chapter of the ACM. In *Berichte des German Chapter of the ACM*, number 4 in Berichte des German Chapter of the ACM, -. Teubner-Verlag.

[Schoenberger und Hofmann, 2014a] Schoenberger, A. und Hofmann, K. (2014a). 3DMemory: Memory usage analysis tool. In *Embedded Computing (MECO), 2014 3rd Mediterranean Conference on*, pages 81–85. IEEE.

[Schoenberger und Hofmann, 2014b] Schoenberger, A. und Hofmann, K. (2014b). Fast Memory Region: 3D DRAM memory concept evaluated for JPEG2000 algorithm. In *System-on-Chip (SoC), 2014 International Symposium on*, pages 1–4. IEEE.

[Schoenberger und Hofmann, 2015] Schoenberger, A. und Hofmann, K. (2015). Analysis of asymmetric 3d dram architecture in combination with l2 cache size reduction. In *High Performance Computing & Simulation (HPCS), 2015 International Conference on*, pages 123–128. IEEE.

[Schoenberger und Reuter, 2015] Schoenberger, A. und Reuter, M. (2015). plasma with FPU. opencores.org/project,plasma_fpu. Last accessed on Jul 7, 2015.

[Son et al., 2013] Son, Y. H., Seongil, O., Ro, Y., Lee, J. W., und Ahn, J. H. (2013). Reducing memory access latency with asymmetric dram bank organizations. *ACM SIGARCH Computer Architecture News*, 41(3):380–391.

[Strutz, 2005] Strutz, T. (2005). *Bilddatenkompression : Grundlagen, Codierung, Wavelets, JPEG, MPEG, H.264*. Vieweg Praxiswissen. Vieweg, Wiesbaden, 3., aktualisierte und erw. aufl. edition.

[Tanenbaum, 2009] Tanenbaum, A. S. (2009). *Moderne Betriebssysteme*. it : Informatik. Pearson Studium, MÜnchen [u.a.], 3., aktualisierte aufl. edition.

[Thakkar und Pasricha, 2014] Thakkar, I. G. und Pasricha, S. (2014). 3d-wiz: A novel high bandwidth, optically interfaced 3d dram architecture with reduced random access time. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 1–7. IEEE.

[The OpenMP, 2014] The OpenMP (2014). Openmp. openmp.org/wp/.

[USC Viterbi School of Engineering, 2013] USC Viterbi School of Engineering (2013). images. http://sipi.usc.edu/database/database.php. Zuletzt aufgerufen am 20.06.2013.

[Weis et al., 2013] Weis, C., Loi, I., Benini, L., und Wehn, N. (2013). Exploration and optimization of 3-d integrated dram subsystems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(4):597–610.

[Wilkes, 1965] Wilkes, M. (1965). Slave memories and dynamic storage allocation. *Electronic Computers, IEEE Transactions on*, EC-14(2):270–271.

[Willenberg und Chow, 2013] Willenberg, R. und Chow, P. (2013). Simulation-based hw/sw codebugging for field-programmable systems-on-chip. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE.

[Wilton und Jouppi, 1996] Wilton, S. und Jouppi, N. (1996). Cacti: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688.

[Woo et al., 2010] Woo, D. H., Seong, N. H., Lewis, D. L., und Lee, H.-H. (2010). An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE.

[Wu et al., 2009] Wu, X., Li, J., Zhang, L., Speight, E., Rajamony, R., und Xie, Y. (2009). Hybrid cache architecture with disparate memory technologies. *SIGARCH Comput. Archit. News*, 37(3):34–45.

[Wulf und McKee, 1995] Wulf, W. A. und McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24.

[x264, 2015] x264 (2015). x264. http://www.videolan.org/developers/x264.html. v2.245.

[Xilinx, Inc., 2010] Xilinx, Inc. (2010). *Memory Interface Solutions, User Guide*.

[Xvid, 2010] Xvid (2010). Xvid. https://www.xvid.com. v2.099.

[Zeigler et al., 2000] Zeigler, B. P., Kim, T. G., und Praehofer, H. (2000). *Theory of Modeling and Simulation - Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, Amsterdam, Boston, 2 rev ed. edition.

[Zhang und Li, 2009] Zhang, W. und Li, T. (2009). Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 101–112. IEEE.

[Zhao et al., 2006] Zhao, W., Belhaire, E., Mistral, Q., Chappert, C., Javerliac, V., Dieny, B., und Nicolle, E. (2006). Macro-model of spin-transfer torque based magnetic tunnel junction device for hybrid magnetic-cmos design. In *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, pages 40–43. IEEE.

[Zhu et al., 2013] Zhu, Q., Akin, B., Sumbul, H. E., Sadi, F., Hoe, J. C., Pileggi, L., und Franchetti, F. (2013). A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pages 1–7. IEEE.

[Ziegler, 2015] Ziegler, C. (2015). Implementierung einer Routine zur flexiblen Verteilung von parallelisierter Software an MIPS-Kernen. Technical report, TU Darmstadt, Integrierte Elektronische Systeme.